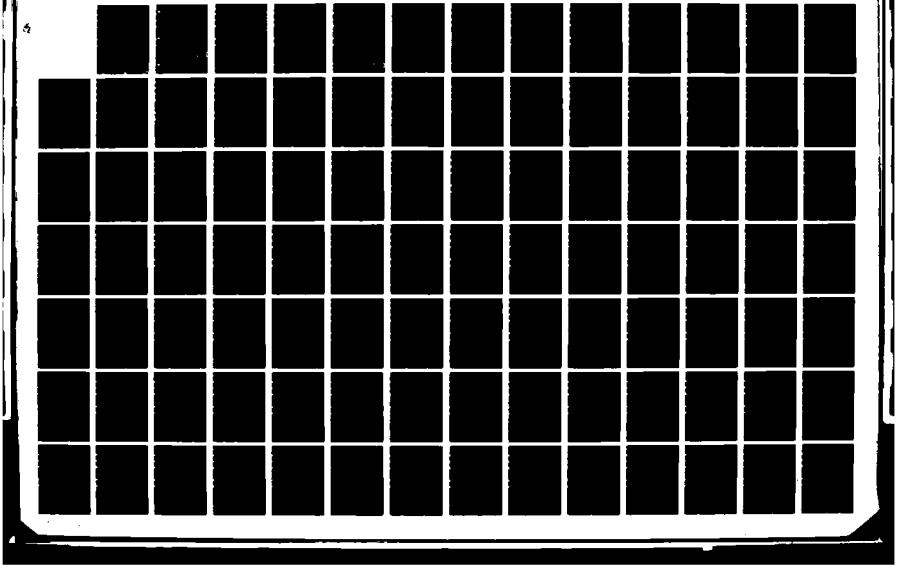
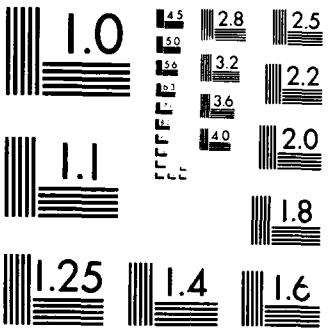
AD-A189 628 COMMON DATABASE INTERFACE FOR HETEROGENEOUS SOFTWARE ENGINEERING TOOLS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 1/3

UNCLASSIFIED T D CONNALLY DEC 87 AFIT/GCS/ENG/87D-8 F/G 12/5 NL





COMMON DATABASE INTERFACE FOR HETEROGENEOUS SOFTWARE ENGINEERING TOOLS

THESIS

Ted D. Connally
Captain, USAF

AFIT/GCS/ENG/87D-8

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

UNCLASSIFIED
RITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/87D-8	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (*If applicable*) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (*City, State, and ZIP Code*) Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6583	7b. ADDRESS (*City, State, and ZIP Code*)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION OSD/SDIO 8b. OFFICE SYMBOL (*If applicable*) S/BM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) Pentagon, Wash DC 20301-7100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

TITLE (*Include Security Classification*)
COMMON DATABASE INTERFACE FOR HETEREOGENOUS SOFTWARE ENGINEERING TOOLS (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Ted D. Connally, B.S., Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1987 December	230

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
Database Management Systems;Programming(Computers); Computer Files;Information Transfer;Interfaces;

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Thesis Chairman: Dr. Thomas C. Hartrum
Associate Professor of Computer Engeineering

Approved for public release: IAW AFR 190-1.
WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ OTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Thomas C. Hartrum	22b. TELEPHONE (*Include Area Code*) (513) 255-3576 / 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

The project involved the design and implementation of a common database interface to integrate a set of heterogeneous software engineering tools. These tools are implemented on a variety of computer workstations, use incompatible data files, and provide little or no database support. The lack of database support and data sharing prevented having an integrated software design environment.

The emphasis of this research was placed on implementing a fully functional database interface which integrated the existing tools and supports the addition of new tools as they become available. The approach selected to implement the interface was the use of a standard data file and a data manager. The standard data file supports all data transfer and the data manager provides all database transaction support.

The unique aspects of the interface is the ability of the standard data file to support multiple tools and the data manager's use of a generic data definition table. The standard data file design addresses the issue of providing a flexible file format which can be modified to support different tools. The use of the data definition table is the key mechanism which allows the data manager to manipulate the data files from various tools. The data definition table describes the contents of the file and supports the data manager in performing database updates and retrievals.

The interface was fully implemented and successfully integrated the existing tools. A tool, developed in a separate research effort, was also successfully integrated with the other tools which demonstrated the interface's ability to incorporate new tools.

AFIT/GCS/ENG/87D-8

COMMON DATABASE INTERFACE FOR HETEROGENEOUS SOFTWARE ENGINEERING TOOLS

THESIS

Ted D. Connally
Captain, USAF

AFIT/GCS/ENG/87D-8

Approved for public release; distribution unlimited

AFIT/GCS/ENG/87D-8

COMMON DATABASE INTERFACE FOR HETEROGENEOUS SOFTWARE ENGINEERING TOOLS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Information Systems

Ted D. Connally, B.S.
Captain, USAF

December 1987

Preface

This report documents my efforts to design and implement a database interface which integrates the software engineering tools available within the Air Force Institute of Technology's Software Engineering Laboratory (SEL). My goal for this thesis was to implement a fully functional interface which not only integrated the existing tools but allows for future tools to be incorporated into the SEL environment.

The interface was successfully implemented using a standard data file to transfer data between the tools and a data manager which performs all database transactions. Hopefully, the implementation of this interface will support and encourage the development of new software engineering tools for use within the SEL.

I wish to express my sincere appreciation to Dr. Thomas C. Hartrum, my thesis advisor, for all his assistance and guidance throughout this effort. I also wish to thank my committee members, Capt Mark Roth and Capt James W. Howatt, for their contributions to this thesis. I would also like to recognize Capt Steven E. Johnson for his contributions in helping me test the database interface using the SADT tool he developed.

Ted D. Connally

Table of Contents

The following additional thesis volumes are maintained at the Air Force Institute of Technology, Department of Electrical and Computer Engineering.

Point of Contact: Dr. Thomas C. Hartrum

Volume II: Data Manager Code

Volume III: Data Dictionary/Data Manager File Translator Code

List of Figures

Abstract

The project involved the design and implementation of a common database interface to integrate a set of heterogeneous software engineering tools. These tools are implemented on a variety of computer workstations, use incompatible data files, and provide little or no database support. This lack of database support and data sharing prevented having an integrated software design environment.

The emphasis of this research was placed on implementing a fully functional database interface which integrated the existing tools and, most importantly, supports the addition of new tools as they become available. The approach selected to implement the interface was the use of a standard data file and a data manager. The standard data file supports all data transfer and the data manager provides all database transaction support.

The unique aspects of the interface is the ability of the standard data file to support multiple tools and the data manager's use of a generic data definition table. The standard data file design addresses the issue of providing a flexible file format which can be modified to support different tools. The use of the data definition table is the key mechanism which allows the data manager to manipulate the data files from various tools. The data definition

table describes the contents of the file and supports the data manager in performing database updates and retrievals.

The interface was fully implemented and successfully integrated the existing tools. A tool, developed in a separate research effort, was also successfully integrated with the other tools which demonstrated the interface's ability to incorporate new tools.

COMMON DATABASE INTERFACE FOR HETEROGENEOUS SOFTWARE ENGINEERING TOOLS

I. Introduction

Background

Within the Air Force Institute of Technology (AFIT), an on-going effort exists to develop software design tools to support a software designer through the phases of the software development life cycle. As part of this effort, a series of theses (4, 16, 17) have produced a set of software engineering tools to form System 690. System 690 provides a design environment for use within the AFIT Software Engineering Laboratory (SEL). The tools run on separate workstations, and are connected via a local area network and modems to a mainframe computer which supports a relational database management system (see Fig. 1).

The primary tool within System 690 is a data dictionary editor (4) which supports the Requirements, Design, and Implementation phases of the software development life cycle. This editor currently runs on Zenith Z-100 workstations and provides the designer a screen-oriented data dictionary editor. A Structured Analysis and Design Technique (SADT, trademark of SofTech, Inc.) Editor (17) is also available which runs on a Sun-3 workstation. This tool provides an interactive graphics editor for SADT diagrams.

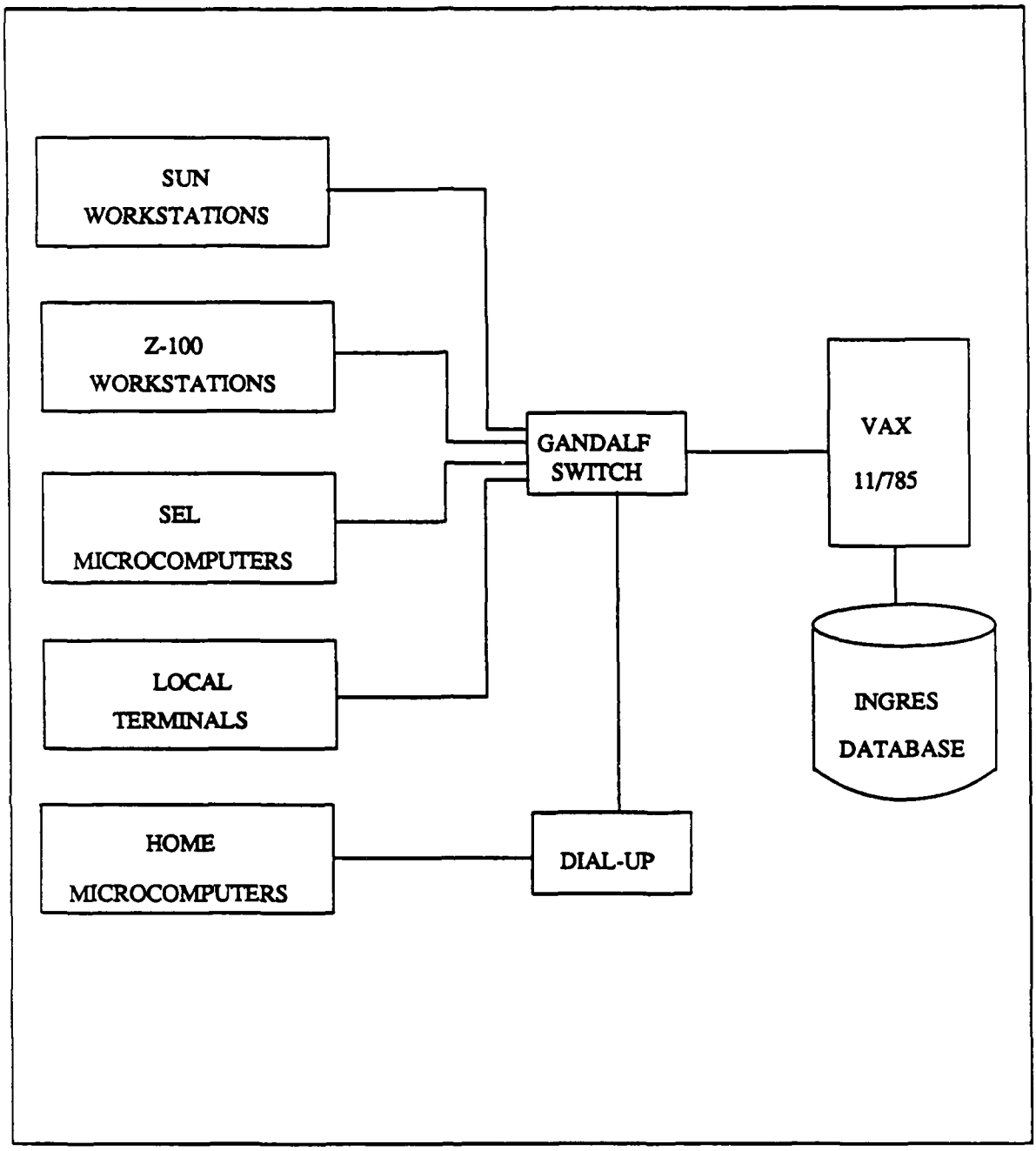


Figure 1. Current Software Engineering Configuration

The goal of System 690 is to provide an integrated system in which a designer could sit down at a workstation, download the necessary data from a central database, work on

a portion of the design, and when finished, upload the modified data back to the database. This data, when stored in a comprehensive, centralized database, would provide a system which could share data between tools and provide the means to document a software project throughout its entire life cycle.

However, a data incompatibility problem exists. The current tools each use a different format for their data files and only a few of the tools interface with the database management system. This data incompatibility problem is being compounded as new tools, using other file formats, are added to System 690. The inability of current and future tools to store and share data prevents having an integrated software development environment within the SEL. Solving the data incompatibility problem would provide two major benefits: an integrated system for use by AFIT students and faculty and a design methodology which could be applied to similar problems in other software development organizations.

PROBLEM

The main problem existing in System 690 is the inability of the separate tools to use a common database and to share data. Implementing a data manager which provides a common interface between the different tools and a central database was the thrust of this effort. The data manager

had to be adaptable so that it could incorporate changes to existing tools and support the addition of new tools.

Scope

The thesis effort covered two specific areas:

1) Analysis of the existing tools and database to design a standard data file format enabling both current and future tools to share a common database.

2) Based on this analysis, the design and implementation of a data manager providing an interface between the tools and the common database.

The design of the standard data file format and the development of a working data manager were the primary areas of emphasis. The ability to add new tools to the system was the main design consideration, with the development of a working data manager which integrates the current System 690 tools being the primary objective.

The computer resources available within the SEL dictated the configuration used to implement the data manager, central database, and standard data file. The data manager was implemented on a VAX 11/785 computer, using the Berkeley 4.3 Unix operating system. The central database was developed using the Ingres relational DBMS. The use of Ingres required that the data manager be developed using the "C" programming language because "C" is the only language available on the VAX system which supported embedded queries with the database. The queries were performed using the Embedded Query Language (EQUEL) provided with Ingres. The

standard data file had to be in a format which was compatible with the Unix and MS-DOS operating systems. This is to support the transfer of the standard data file between the System 690 tools and the data manager.

Assumptions

Two assumptions were made in the development of the data manager. The first assumption was in determining the primary communication interface to support between the tools and the data manager. Because the data manager resides on a central computer and the bulk of the tools reside on individual workstations, the communication interface focuses on supporting remote accesses, either modem or local area network. The second assumption was that a System 690 tool designer either has a sufficient database background to identify the contents to use in the data manager control relations or can obtain the necessary assistance to identify them.

Approach and Presentation

The thesis effort was carried out in the following phases:

Phase 1 -- Reviewed the existing System 690 configuration and current literature to determine system requirements.

Phase 2 -- Requirements analysis of the standard data file format and data manager.

Phase 3 -- Design of the standard data file format and data manager.

Phase 4 -- Development and implementation of the standard data file format and data manager.

Phase 5 -- Measurement of data manager performance.

Chapter II contains the review of the current system configuration and literature. The review examined the current System 690, the database functions other systems identified as important, and techniques used to integrate heterogeneous tools. The System 690 review concentrated on the features, capabilities, and requirements of its tools. Other integrated environments were reviewed to identify the database function requirements necessary to provide a sound environment. Finally, the methods used by other systems to integrate stand-alone tools were examined.

The requirements analysis is in Chapter III. The analysis established the requirements for the data manager and the standard data file. The most important requirement established was the need to be able to build the standard data file using a "generic" means to describe the standard data file's contents for each of the tools. The data description needs to support both updates and retrievals to reduce maintenance overhead and prevent possible errors arising from having the same basic data maintained in two places.

Based on the requirements analysis, the standard data file and data manager were designed. The design analysis consisted of both a system design and a detailed design.

The system design in Chapter IV presents the system design selected for the data manager, standard data file, and their configuration. The detailed design of the standard data file, data manager, and its components are in Chapter V. The design concentrated on providing a system which can easily incorporate new tool data file formats as the new tools are added to System 690.

Chapter VI presents the implementation features and testing procedures followed in implementing the data manager. This phase was programming intensive with the objective of obtaining a working system. Chapter VI also contains an evaluation of the data manager's performance and its ease of integration with software engineering tools.

The final chapter summarizes the results of the thesis and provides suggestions for future enhancements to the data manager and the System 690 environment.

II. Review of AFIT Environment and Literature

Introduction

The main work done in this thesis was the implementation of a data manager which provides a generic interface between System 690 tools and a central database. To gain a better understanding of the problem and its solution, a review was made of the current system and literature to examine the work others have done in similar efforts.

There are three topics which provide a strong insight into this thesis. The first topic is a review of the work accomplished in previous thesis efforts which produced the current System 690 configuration. The next topic is an examination of the database functions which the data manager needed to address. The final area studied is the methodology other systems have used to integrate non-homogeneous tools.

System 690 Configuration

Before examining other systems and how they address problems similar to those of this thesis, the current System 690 tools and their development are reviewed. The objectives the tools are trying to achieve, the tools themselves, and their configuration are discussed.

The basic objective of System 690 is to support the standard software development methodology established in the Software Development Documentation Guidelines and Standards

(5). These guidelines establish a standard documentation guide which can be used for an entire software development project (5: 2). The method used to support this standard is a data dictionary. A dictionary entry is established for the requirements, design, and implementation phases of the software life cycle. Each of these phases consists of a set of action entities and a set of data entities for a total of six types of dictionary entries. Refer to Figure 2 for a sample data dictionary entry.

Several thesis efforts have produced a set of automated tools to support the concepts set forth in the Software Development Documentation Guidelines and Standards (5). Thesis efforts by Thomas (16) and Foley (4) have provided an automated data dictionary (DD) editor. The DD editor addresses all three phases (4) established in the Guidelines and Standards. Urscheler's thesis (17) produced an interactive graphics SADT editor.

Thomas' thesis provided one key component to System 690: a data dictionary database (6: 652). Thomas developed the database schema for all the dictionary entries (16: 84-147). He also implemented an interactive DD editor which interfaced with the database (16: 166-182).

Thomas' DD editor was implemented on a VAX 11/780 running the UNIX operating system (16: 184). Because of the UNIX environment, Ingres was selected as the DBMS for the

tool (16: 185). This configuration established the environment Foley used in his work.

```
NAME: mess_parts
PROJECT: NETOS-ISO
TYPE: PARAMETER
DESCRIPTION: Decomposed message parameters.
DATA TYPE: Composite, probably C structure or PASCAL record.
MIN VALUE: None
MAX VALUE: None
RANGE OF VALUES: None
VALUES: None
PART OF: None
COMPOSITION:    SRC
                DST
                SPN
                DPN
                USE
                QTY
                Buffer
ALIAS: Message Parts
   WHERE USED: Passed from Decompose Message to Validate Parts
   COMMENT: Part of earlier design
ALIAS: messy-parts
   WHERE USED: Passed from Dump Data to Flush Buffer.
   COMMENT: Part of existing library.
REFERENCE: MSG_PARTS
   REFERENCE TYPE: SADT
VERSION: 1.2
VERSION CHANGES: Component USE added to allow network messages
DATE: 11/05/85
AUTHOR: T. C. Hartrum
CALLING PROCESS: Process Message
   PROCESS CALLED: Decompose_Message(parts list)
   DIRECTION: up
   I/O PARAMETER NAME: parts_list
CALLING PROCESS: Process Message
   PROCESS CALLED: Process Network 4 Messages
   DIRECTION: down
   I/O PARAMETER NAME: parts
```

Figure 2. Sample Data Entity Entry in Design Phase (5: 29)

Foley's thesis was an enhancement to Thomas' work. Thomas' tool was implemented on a heavily used system within AFIT and was slow enough to generate complaints from its users (4: A-4). To address this issue, Foley developed a microcomputer (Z-100) based DD editor which allowed the user to perform the bulk of his work without directly interacting with the database (4: 45-72).

Foley's DD editor implemented a forms based editor for each of the phases but only implemented a prototype database interface for the design phase data (4: 70). This interface converts the DD editor tool file into database entries and vice-versa (4: 70). Unfortunately, this code is highly specialized for this phase and extensive work would be required to extend the interface for the other phases.

Foley's thesis provided two key components to System 690. First, the Z-100 based DD editor provided a tool which was significantly more "user friendly" (4: 90) than Thomas' tool and the additional work on the design phase further refined the database schema (4: 35). Refer to Figure 3 for the schema of the data entity in the design phase.

While Thomas and Foley developed a text-based means to create data dictionary entries, Urscheler developed a graphics-based SADT editor for use within the requirements phase (17). A SADT diagram (see Figure 4) represents one level of a hierarchical decomposition of a system's func-

tions (13: 192). The SADT editor allows a user to edit and manipulate an entire diagram (level) at a time.

parameter	
project	c12
paname	c25
datatype	c25
low	c15
hi	c15
span	c60
status	c1

padesc	
project	c12
paname	c25
line	i2
description	c60

paalias	
project	c12
paname	c25
aliasname	c25
comment	c60
whereused	c25

pahistory	
project	c12
paname	c25
version	c10
date	c8
author	c20
comment	c60

papassed	
project	c12
paname	c25
prcalling	c25
prcalled	c25
direction	c4
iopaname	c25

pavalueset	
project	c12
paname	c25
value	c15

pahierarchy	
project	c12
hipaname	c25
lopaname	c25

paref	
project	c12
paname	c25
reference	c60
reftype	c25

Figure 3. Database Schema for a Data Entity Within the Design Phase (4: 37)

As part of this tool, Urscheler intended to implement an interface with the DD editor database to store the data dictionary information generated by his tool (17: 23) but the interface was not completed (17: 43). Further research (9), occurring in conjunction with this thesis, is enhancing

the functionality of the SADT editor by providing a more complete set of the SADT language. The research is also developing the means to generate and use data dictionary information. The enhanced SADT editor is being designed to use the standard data file generated by the data manager developed in this thesis.

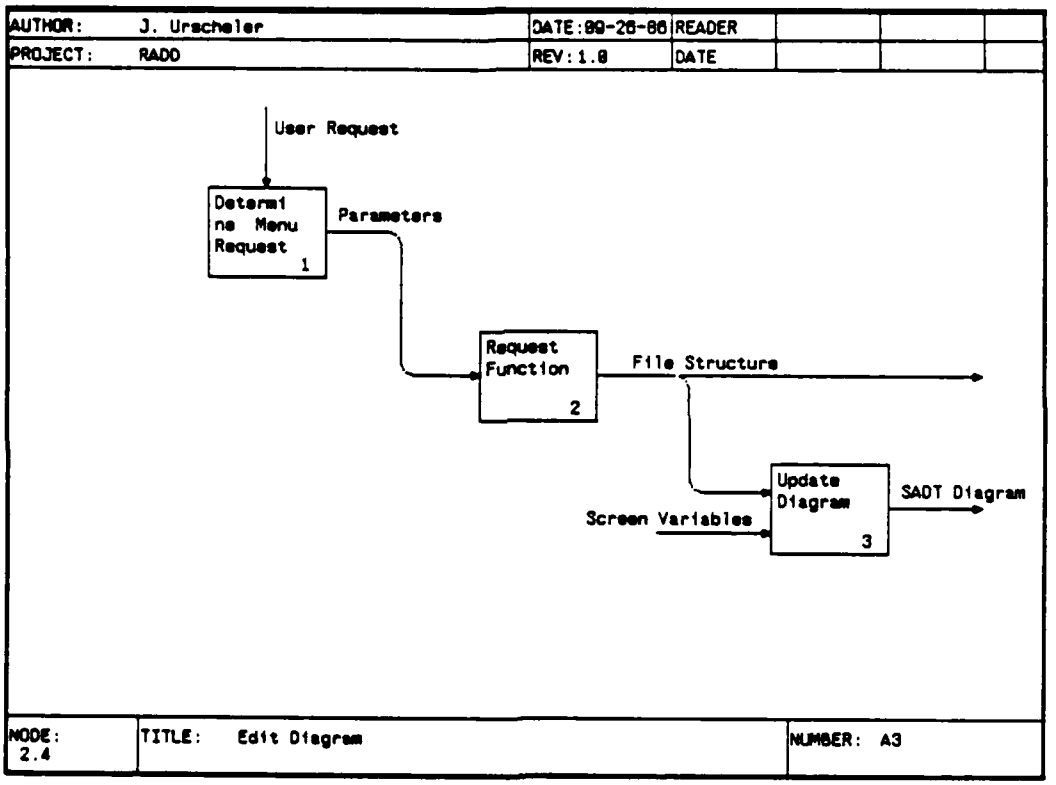


Figure 4. Sample SADT Diagram (17: B-8)

The DD editor and SADT editor are connected to a VAX 11/785 to create the current System 690 configuration. As shown in Figure 5, there is an excellent opportunity to create an integrated environment where all the tools share data via the Ingres DBMS. However, prior to this research,

only the design phase of the DD editor could interface with Ingres. This prevents the tools from sharing information and automated consistency checkers cannot be used against the various phases of a design (6: 652). This inability of tools to use and share a common database is the main problem this thesis addresses.

Database Functions

In order to support System 690 tools, the data manager is responsible for many functions that the Ingres DBMS system does not provide. To gain a better insight into what these functions are, different software engineering environments (SEE) were reviewed to determine the basic goals of a SEE. The focus was then placed on the functions a SEE's data manager had to provide.

Before beginning the discussion, one important point must be made. The literature brought out a relationship between SEE and CAD/CAM database requirements. Randy Katz indicated this relationship by grouping large software systems and integrated circuit designs into the category of "complicated engineering artifacts" (12: 191). This relationship proved useful in the examination of database functions because of the large amount of research occurring in the CAD/CAM database area.

Software Engineering Environments. There are a number of SEEs currently under development, but the two most applicable to this thesis are ARGUS (15) and SODOS (7).

Both of these systems provide support throughout the software life cycle (SLC), whereas other systems tended to focus more on the coding phase and the associated tools, ie. text editors, compilers, debuggers, etc.

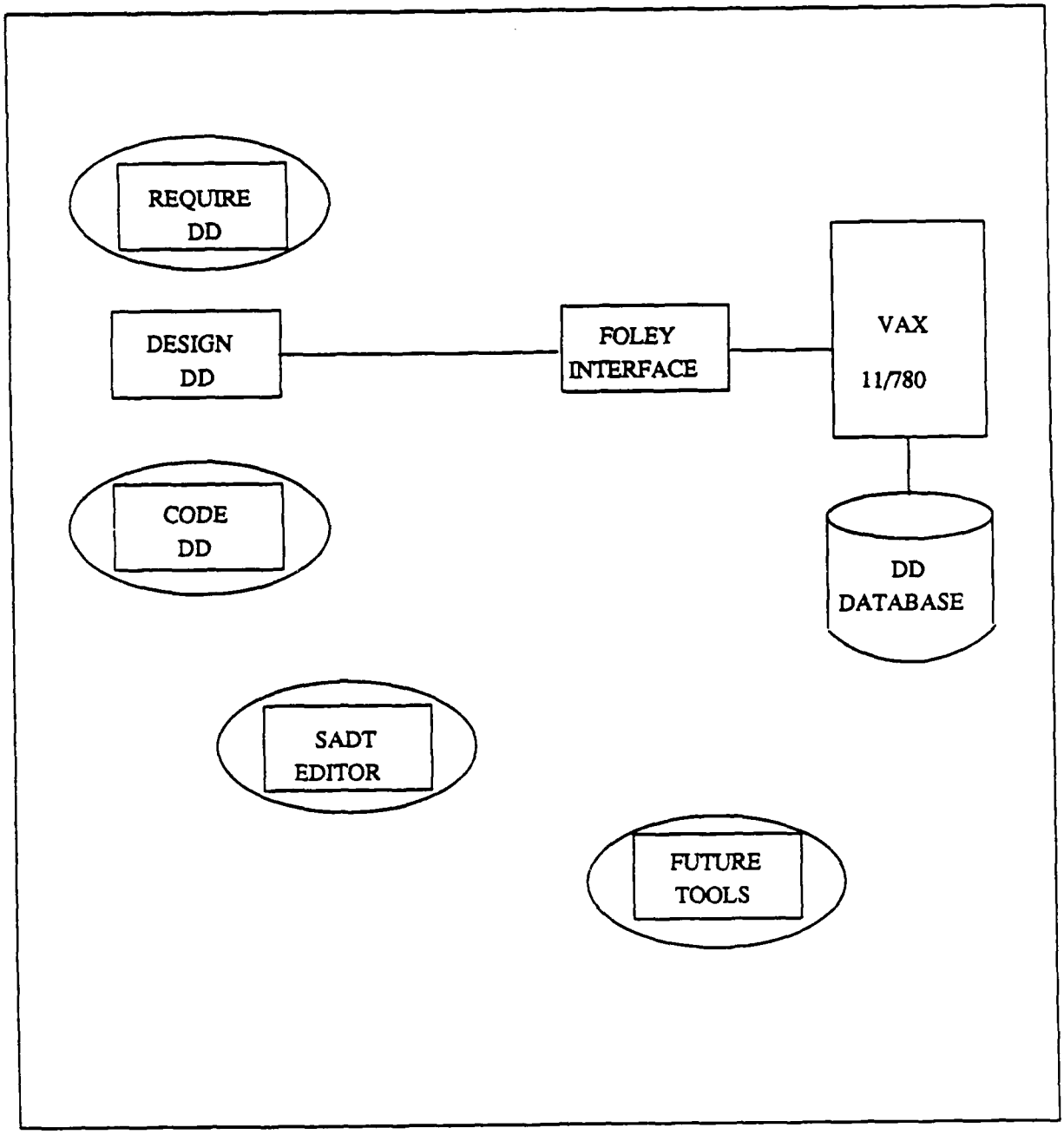


Figure 5. Current System 690 Configuration

Before reviewing these two systems, a brief overview of general SEE features is presented. These features are extracted from a survey article (3), which summarized the general features a SEE should have. Its author points out that these features are not absolute for all systems (3: 457) but they reflect the goals of both ARGUS and SODOS. The suggested SEE features are that the SEE:

1) support the entire life cycle,

2) allow links between phases of the life cycle, both forward and back,

3) contain a consistent interface,

4) contain a project/software/group database,

5) support project management,

6) enforce configuration management,

7) be expandible/flexible,

8) contain powerful tools, which are integrated and automated,

9) have reusable tools; facilitate reusability of software, and

10) be portable (3: 457).

System 690 addresses most of these features but the key objectives for this thesis are 1-4, 7, and 8. The data manager needs to support each of these features. The first four topics are covered with respect to the ARGUS and SODOS systems. The flexibility and integration issues are addressed in the following section of this chapter.

The ARGUS system is a comprehensive software engineering environment using "CAD/CAM-like" principles for software

development (15: 129). It shares many of the same objectives indicated above. The primary functions of ARGUS are to

> "... provide computer assistance with the specification, analysis, construction, and maintenance of various software products throughout the total life cycle. Following a CAD/CAM-like approach, this semi-automated system assists the user in capturing and controlling design configurations and tracing these specifications throughout the entire lifecycle (15: 130)."

ARGUS supports the full range of the software life cycle. It breaks the life cycle down into phases and provides "toolboxes", containing phase specific tools, to support the user within a particular phase (15: 131-132). These toolboxes are Manager, Designer, Programmer, Verifier, and General. The Designer Toolbox is the most relevant to System 690 because it contains the tools used for documenting and controlling the formal design of a software system.

The Designer Toolbox is the portion of ARGUS which best utilizes its CAD/CAM functions. It controls both graphic and textual data in its support of requirements analysis and design (15: 133). It provides templates to the user for each of the reports used within each phase of the software life cycle. Its use of a relational database to control the data allows ARGUS to maximize its information leverage. This leverage is provided by storing all data in a central database where it can be projected to any designer.

The power provided by the relational database makes the database a key part of the ARGUS system (15: 129, 133). One

of the most powerful aspects of the database is its capability to maintain only one copy of the data and project this data to tools based on an appropriate template. This concept has yielded a data compression ratio of nearly 1 to 4. This concept provided the basis for the data manager design and shows the possible benefits to be derived.

The SODOS (Software Documentation Support) system is another SEE which supports the entire Software Life Cycle (SLC) (7: 8). Its emphasis is on supporting the definition and manipulation of software development documentation. SODOS has two main objectives. One is to provide maintenance personnel all the information generated during the specification and development phases, and the other is to help system developers generate the necessary documentation with a minimum of extra effort.

SODOS is based on the use of a relational DBMS and stores all data in the project database (7: 8). The data generated in each phase is inter-related based on a set of pre-defined relationships. A model is used to identify the information in the SLC, the relationships among the information, and how it is used.

The data for each phase is stored in accordance with a document definition which allows the document administrator to define new documents based on a database schema (7: 9). This schema establishes the document structure, inter-relationships, keywords, and related documents. This format

provides a large amount of flexibility from project to project. As part of this definition capability, the relationships between the data in the separate phases are established, which allows easier consistency checking.

The SODOS system exhibits many of the characteristics of the current data dictionary editor used in System 690. Both systems address the requirements, design, and implementation phases of the SLC. The concepts shown in the SODOS system point to the feasibility of this thesis effort and its contribution to providing a more useful environment.

Data Manager Functions. The surveyed environments all utilized some type of data manager to interface with a DBMS to store and manipulate the data. These data managers are pivotal to the success of SEEs and, as stated earlier, to the success of CAD/CAM systems. The bulk of the research performed in defining a data manager's functions is in the CAD/CAM area, especially in VLSI design. For this reason, much of the following is extracted from CAD/CAM environments but is applicable to SEEs as well.

A good summary of the requirements and problems a data manager must address was developed by G. P. Barabino and others (1). The requirements and problems are the

1) management of complex data schemata,

2) likelihood of frequent changes in data organization,

3) manipulation of huge amounts of data,

4) control of data coherency and redundancy minimization,

5) security against unauthorized accesses,

6) provisions for back-up and crash recovery procedures,

7) support of concurrent access to data by many designers,

8) support of design administration and project management,

9) automatic enforcement of some consistency constraints on data,

10) support of design hierarchy and complex objects,

11) use of long fields for storage and retrieval of unformatted information,

12) provisions for navigational facilities among design data,

13) support of long lasting transactions, and

14) interactive level performance (1: 577).

The above requirements are in agreement with other authors except for managing versions, which is not listed. Barabino does address versions in another article (2: 800) which improves his approach.

As Barabino points out, the first seven requirements can probably be handled by current relational DBMSs (1: 577). However, the remaining functions require either an extension to the DBMS or a data manager which works with the DBMS. The Barabino effort used the latter approach, developing a means to satisfy the requirements without modifying the DBMS. The system utilizes three interfaces:

1) Direct access with the DBMS (Ingres)

2) A Data Base Interface (DBI) Module which supports the requirements listed in items 7 - 12

3) A Local Information Processing Subsystem (LIPS) which provides an interface between the applications and the DBMS for limited design data subsets requiring short response times. (1: 578)

The DBI is important because it provides several functions which are similar to those required in this thesis. The DBI is implemented using EQUEL (Embedded QUEL) and supports project management, consistency constraints, design hierarchies, data management, and navigation through the data (1: 578). According to the authors, it allows the data management and application program problems to be addressed separately. By doing so, the DBMS structure can be changed without changing the application and vice-versa. The DBI also addresses the management of design versions and alternatives (2: 800).

The functions a data manager are to provide have also been addressed by Randy Katz (11, 12). Katz's considerations are similar to Barabino's (1) with the addition of the following:

1) A design librarian supporting check-in/check-out of design parts from the database

2) A method to track design versions (11: 27).

The design librarian is an important component because it coordinates all access to shared design data in the central database (11: 33). By employing proper check-out policies, it guarantees only one in-progress version exists

of an object. This control, allowing the objects to be viewed but not updated, improves the usability of the system.

The means to manage design versions is also a vital component within a design environment. Katz feels that proper support of versions is critical for successful design data management (12: 192-193). Two of the primary goals he feels must be met are for the versions to require minimal redundancy and be quickly retrievable. Katz presents several approaches for version management which the interested reader may find discussed in detail in 12: 193-200.

Tool Integration

The ability to integrate heterogeneous tools is becoming a key research concern as more and more such systems reach the user's market (10: 111). This is especially true as evidenced by this definition of a SEE:

> "... a set of tools, structures, rules, and procedures that together provide a framework for software development and support (3: 456)."

The key part of this definition is that the tools and structures work together. Since a goal of this thesis is to provide a means to integrate System 690 tools, the methods used to interface different tools are examined.

There are several methods used to interface different tools. The most common are

1) ad hoc communication between each pair of tools using pre- and post-processors,

2) placing all the applications within one environment, and

3) using a single database manager with each application interfacing with the data manager to access the data stored in a DBMS (10: 112-113).

Of these, the third is most relevant to this thesis. Ad hoc communications are not considered because of their inefficiencies (10: 112). Placing all tools in one environment is not relevant given the current configuration and goals of System 690.

A data manager has been implemented in systems such as Polylith (14) and SDE (8). The basic concept of these systems is to provide a single interface between the tools and the central database. Without this single interface, each tool would have to manage a "power set" number of interfaces to communicate with other tools (14: 13). The efficiency and applicability of this approach to the System 690 data manager makes these concepts especially relevant.

The Polylith system tries to join various component tools by addressing the problems of data interchange and tool synchronization (14: 12-14). The system uses a Polylith grammar, in conjunction with a Message Handler, to provide the tool interface. The Polylith grammar defines the data and format required for a tool and the Message Handler uses this information to retrieve and build the appropriate message for the tool.

The Message Handler provides the data manager role in the Polylith system (14: 13-17). It carries out all

transactions requested by the tools and tracks the status of the data. The author identified the Message Handler's ability to perform automatic data transformation as a novel aspect of the Polylith system. The biggest advantage of this capability is that the system allows the addition of new tools to the system without having to discard or modify old tools.

The SDE system offers a similar approach to the tool integration issue (8). Hsu addresses the problem of interfacing design tools which were originally designed to be stand-alone systems (8: 733). The SDE integrates the system's various tools.

The data manager is the kernel of the SDE and is based on the Ingres DBMS (8: 734-735). It controls all access to the data, defines the data structure, and provides a common database for the various designers. It also manages version and configuration control.

Although the data manager is vital to the SDE, a tool manager is required to perform the actual tool integration (8: 735-736). A basic data structure was identified which was common to all tools. This structure is the basis for the tool integration. The tool manager adds tool specific views and information to the basic structure to allow the tools to interface. Once the tool receives the data, it can operate in a stand-alone mode, and upon completion, load the new data into the common database via the tool manager.

The key features observed in the Polylith and SDE systems are their use of a common data structure for messaging and a manager which can manipulate this structure to support the requesting tool. The key advantages observed were the reduction in the number of interfaces a tool had to maintain and the adaptability of the system to incorporate new or modified tools with minimal system disruption.

Summary

The review of System 690 environment showed it to consist of two primary tools; the DD editor and an SADT editor. While both operate well in a stand-alone mode, they currently have limited or no access to a central database. Because of the continued work with these tools, the database schema has been identified for their inclusion in the central database. The basic problem is how to interface these tools with the central database.

The review of current literature examined other environments and research efforts for an insight of how they addressed problems similar to those of System 690. Their solution was to implement a data manager which used a standard interface for all tools. The tools interfaced only with the data manager and did not worry about formatting data for another tool's use.

The findings of this review show that this thesis addresses items which are current, relevant concerns of the software engineering community. Further, the review iden-

tified many of the requirements of an integrated system and the role the data manager is to play. These findings will be beneficial in the following Requirements Analysis chapter.

III. REQUIREMENTS ANALYSIS

Introduction

The basic objective of this thesis was to provide a common interface which integrates the tools within System 690, enabling them to use a common database and share data (Fig 6). A review of the current SEL and research efforts addressing similar situations showed that a data manager controlling the database and using a standard data file as the interface between the tools and the data manager was a valid approach (1, 8, 10, 14). An important component also identified to support an integrated environment was a means to provide a database librarian function through session control (2, 11, 12).

This chapter provides an overview of the system's basic operations and then establishes the requirements of the standard data file, data manager, and the data manager's session control mechanism. The requirements of the common database are also examined.

Before beginning the requirements analysis, several definitions are needed. A data entity refers to all the information describing a data dictionary entry. The data entity consists of multiple data elements. These data elements are the values representing specific data fields in a data dictionary entry. A session refers to a tool-data manager interaction where data is retrieved from the

database, manipulated by the tool, and stored back into the database. A transaction is a request to the data manager to perform a database retrieval or update.

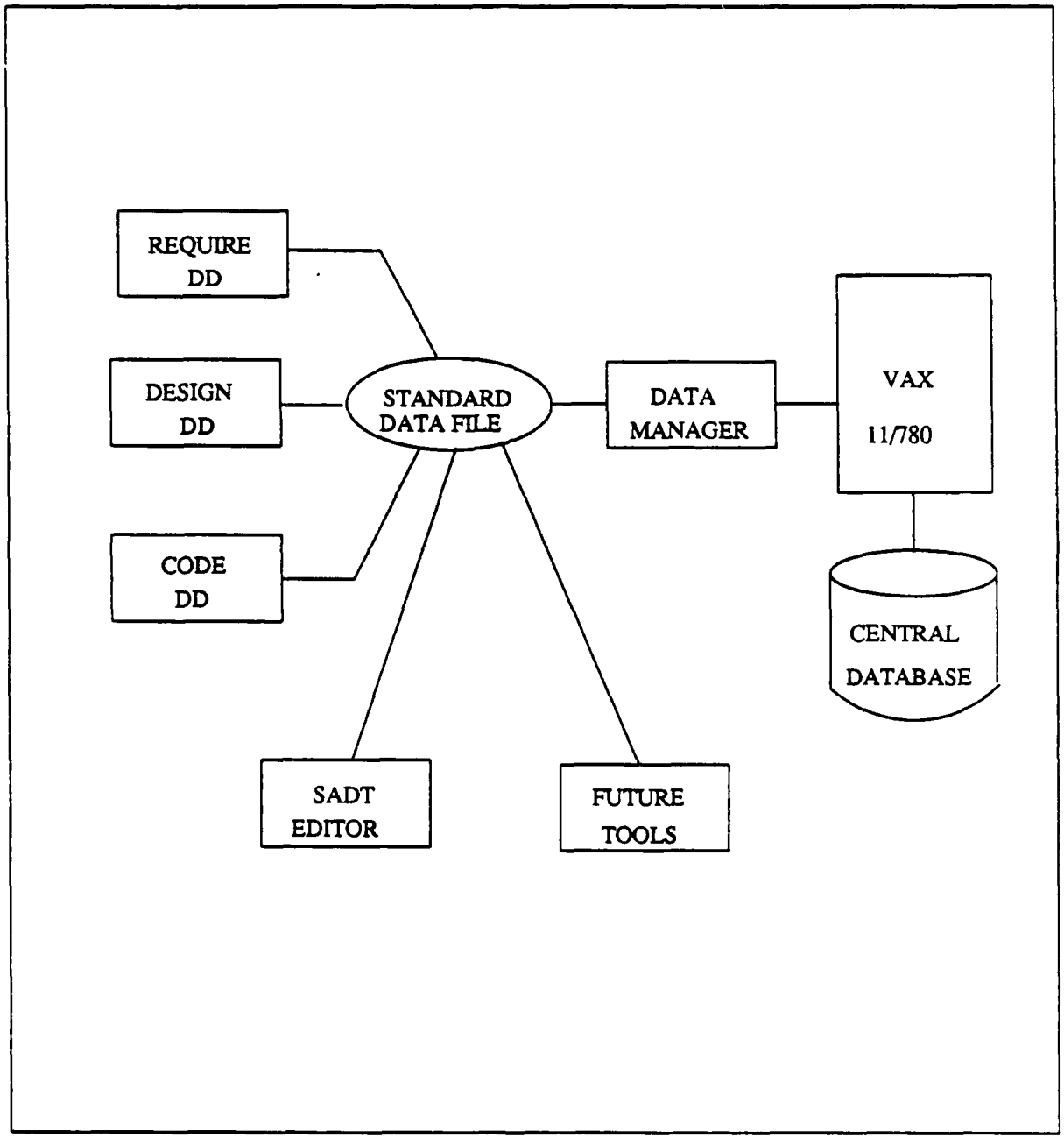


Figure 6. Goal System Configuration

Overview

The primary requirement of this thesis is to provide a standard data file and a data manager which can manipulate the standard data file. These two components must be adaptable to changes and additions in tool data needs. The standard data file is the data interface between the SEL tools and the data manager. The data manager must use and create the standard data file and provide session control to meet its requirements. The overall interactions between the data manager, the tools, and the database are shown in Figure 7.

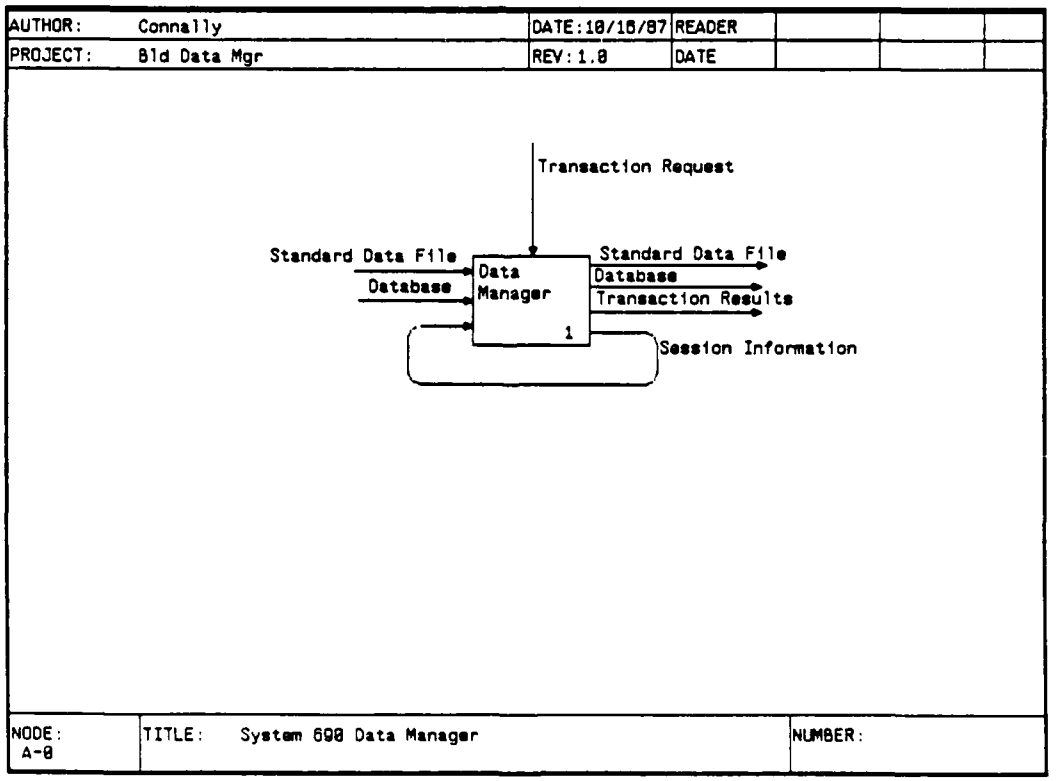


Figure 7. Data Manager SADT Top Level

The data manager performs many tasks but its basic function is to retrieve data from and write data to a common database using the standard data file. This is reflected in Figure 8 which shows the major activities the data manager must perform and the data it is required to use and generate. The following sample session is provided to show how the data manager uses and creates the various data items.

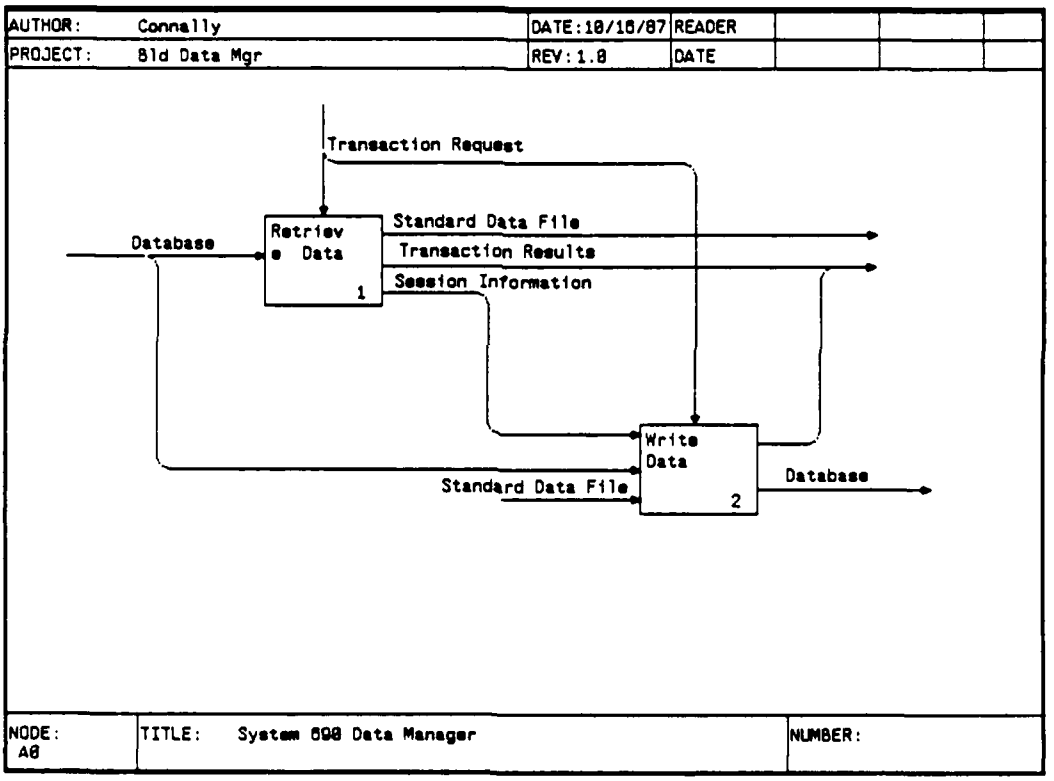


Figure 8. Data Manager SADT First Level

Sample Session: A user or tool will request a data entity(s) from the database. On receipt of the request, the data manager will retrieve the data and provide this data back to the requestor in a standard data file. When retrieving the data, the data manager needs to provide session control to maintain database integrity.

When the desired changes have been made to the data, the tool which checked-out the data, requests to update the database. The data manager will use the standard data file, containing any changes made by the tool, and the session information generated during the retrieval to coordinate and perform the database updates. After the data is successfully written back to the database, the session is terminated.

This sample session shows how a typical session will be performed. It also indicates the dual role the standard data file and session information provide. The impact of this dual usage will become evident in the remainder of this chapter.

Standard Data File

The standard data file is the means used by the data manager to transfer data between System 690 tools and the common database. It provides a standard file structure for all tools to use in interfacing with the data manager. The file is the interface and therefore must contain not only the requested tool data but also provide control information to the tool by describing the contents of the file.

The requirements of the standard file are examined with respect to its two functions: data transfer and file description. The data format requirements are based on the types and structures of the data being transferred. The file description requirements are based on the information necessary to inform a tool and the data manager what the contents and structure of the file are.

Data Format. The initial step taken in determining the data format requirements was to identify the minimum number of file entries necessary to describe a data element to a tool. Then, based on an examination of the data dictionary fields and the relations in the current databases (4, 16), additional entries were included in the data format.

The absolute minimum information needed in the data format is the data itself. Without additional information, the tool would have to depend on positional notation to process the data. This is not a valid option because the number of data elements used in a data dictionary entry varies from one entry to the next. To avoid this, the name of the data element must also be included.

The next entry required is the length of the data element. A data element may be any valid Ingres data value length, therefore no default value may be assumed. Furthermore, one tool may display an element as an 60 position field while another uses it as a 40 position field. To prevent the possible loss or incorrect use of data, the inclusion of an element's length is a minimum requirement.

The minimum entries identified are the name, element length, and value of a data element. However, these entries only describe a data element's characteristics, not its function within a data dictionary entry. To further identify a data element, its function within a data dictionary entry needs to be established. These functions can be

broken down into several basic field types: single-line fields, multi-line fields, group fields, and multi-line fields within a group field.

A single-line field is the simplest of the data structures to be transferred. NAME (Fig. 9) is an example of such a field. The already identified entries adequately describe this data field. Therefore, this field places no additional requirements on the data format.

A multi-line field is not as simple. DESCRIPTION (Fig. 9) shows a single data field which may consist of several lines in a display. The database tracks such an element by lines (reference padesc in Fig. 10). To indicate to a tool it is dealing with this type of data element, a multi-line indicator is required in the data format entries.

A group field also requires an additional entry in the data format. The ALIAS field (Fig. 9), with the associated COMMENT field, is an example of such a field type. The relationship between these data elements need to be reflected in the data format file.

The last type of entry, multi-line field within a group field, is not required by any of the current tools but may be added by future tools. An example of such a field would be the COMMENT (Fig. 9) entry being a multi-line field within an ALIAS entry. The data format would have to be able to identify this type of element to a tool.

The data dictionary field types were used to identify the above requirements. However, the relations used in the database create additional requirements. Certain data dictionary fields use relations where the value of an element and its use in a data dictionary entry are determined by other attributes within the relation. Examples of this type of data dictionary field are the INPUT DATA, INPUT FLAG, OUTPUT DATA, and OUTPUT FLAG fields (Fig. 9). The relation attribute (paname) is the same for each entry and is stored in the processio relation (Fig 10). Since all four fields use the same data value, some means to identify the differences is required. The differences are shown in

```
NAME: (of process)
PROJECT: (project name)
TYPE: PROCESS
NUMBER: (node number of this process)
DESCRIPTION: (Multiple lines allowed)
INPUT DATA: (Multiple lines allowed)
INPUT FLAGS: (Multiple lines allowed)
OUTPUT DATA: (Multiple lines allowed)
OUTPUT FLAGS: (Multiple lines allowed)
ALIAS: (Multiple entries allowed)
   COMMENT: (Why this alias is needed)
CALLING PROCESSES: (Multiple lines allowed)
PROCESSES CALLED: (Multiple lines allowed)
ALGORITHM: (Multiple lines allowed)
REFERENCE: (Multiple entries allowed)
   REFERENCE TYPE: (SADT, text, etc. for this reference)
VERSION: (Version of this data dictionary entry)
VERSION CHANGES: (What was changed from the last version)
DATE: (Of this data dictionary entry)
AUTHOR: (Of this data dictionary entry)
```

Figure 9. Process Data Dictionary Entry (5: 26)

the direction (IN/OUT) and type (FLAG/DATA) attributes in the processio relation (Fig. 10). A similar method is required to uniquely identify the data values in the standard file.

process		prdesc	
project	c12	project	c12
prname	c25	prname	c25
number	c20	line	i2
status	c1	description	c60
processio		pralias	
project	c12	project	c12
prname	c25	prname	c25
paname	c25	aliasname	c25
direction	c4	comment	c60
type	c4		
pralg		prcall	
project	c12	project	c12
prname	c25	prcalling	c25
line	i2	prcalled	c25
algorithm	c60		
prhistory		prreference	
project	c12	project	c12
prname	c25	prname	c25
version	c10	reference	c60
date	c8	reftype	c25
author	c20	comment	c60

Figure 10. Process Database Relations (4: 36).

The combination of the initially identified requirements, the data dictionary field types, and database storage mechanisms have produced the following minimum entries to identify each data element in the standard file:

1) Name

2) Length

3) Contents

4) Multi-line

5) Multi-field

6) Direction

7) Type

File Description. The requirements for the contents of the file description header are based on the need to describe the structure of the data elements to both the tools and the data manager and provide control information to them. The data in the file will need an identifier which indicates the type of transaction which generated the file. To prevent a tool from trying to use data structured for another tool, a means must be provided to insure the data contained in the file is compatible with the tool. A method to show the type and amount of data contained in the file is also needed to describe the total contents of the file to the tool and data manager. The final requirement is a provision for a means to track tool usage data for use in analyzing usage patterns and tool performance.

The identification entry indicates whether the standard data file was generated by a tool or the data manager. A file created by a tool would be used by the data manager to update the database with new data. A file generated by the data manager would be used by a tool to manipulate retrieved data. This file needs a session identifier associated with

it which identifies the contents as being checked-out. The data manager will need this identifier to check the file back in. Because the identifier must be unique to identify each checked-out file, the tools will need to maintain this identifier throughout a session.

A tool identifier must be associated with a file to insure that the tool accessing the file is compatible with the data it contains. A basic requirement of this information is a header entry which uniquely identifies the tool for which the file was generated.

The data description requirements fall into three basic categories: indication of the project name, the phase and type of the data entities contained, and a list of all the data entities in the file.

The project name of entities contained in the standard data file is included in the file description header because of its importance in identifying a data entity. Project serves as part of the key of every data dictionary relation. The addition of the project name provides a more complete description of the standard data file's data entities.

The phase and type information is required for those tools which can manipulate data entities in more than one phase of the software life cycle and/or use data that can be either object or action entities. The phases and data types refer to the phases of the software life cycle and the data types which occur within each phase. The data manager is

required to support three of the life cycle's phases: requirements, design, and implementation. Each phase uses activity and data entities to describe the software system within the phase. The requirements phase's data types are activity and data item. The design phase's data types are process and parameter. The implementation phases's data types are module and variable. For a complete discussion of the phases and data items used in the SEL, refer to the Software Development Documentation Guidelines and Standards (5). For the remainder of the thesis, phase will refer to the requirements, design, and implementation phases. Action entities will refer to the activity, process, and module entities. Object entities will refer to the data item, parameter, and variable entities.

The DD editor is an example of a tool which uses a wide range of data types. It can edit data in all three phases and uses both object and action entities. Without both phase and type information, the data is not adequately described.

A summary of the data entities is necessary for those tools which access multiple data entities within a session. The SADT editor is an example of a tool which uses both action and object entities within a single session. Therefore, a method to identify the name of an entity and whether it is an action or object entity is needed. The

number of entities is also required to support any tools which dynamically allocate memory for their workspace.

An entry is also required which indicates if a data entity is in a read or write status. This provides the tool and data manager information which allows them to implement some type of update control or to highlight data it does not have permission to update. This feature is necessary to inform the user that he is attempting to update data he does not have the privilege to modify.

The final requirement of the file description information is for it to be able to track tool usage. A tool usage monitor is available which measures the time a user spends within a session. The monitor requires the start and stop date and time for each session. Entries to contain these times must be included in the file description header.

The evaluation of the file description header has produced the following requirements:

1) Session Identification
2) Tool/File Compatibility Header
3) Project
4) Phase
5) Type
6) Data Entity Summary
    a) Entity Name
    b) Action/Object Type
    c) Read/Write Status

7) Start/Stop Time Entries

Data Manager

The data manager's requirements cover a broad range of functions. The data manager's primary functions are to retrieve data from and write data to the database. The data manager must also provide an interface which allows tools and users to specify the transactions to be performed. The data manager must generate and use the standard data file. It also needs to support some type of session control to protect database integrity.

Tool/User Interface. The tool/user interface is provided to allow a tool/user to specify to the data manager the type of transaction to perform and provide the tool/user the transaction's results. The interface must support both interactive and batch requests. This requirement provides greater flexibility for smart tools that can build batch requests without the user having to interface directly with the data manager (Fig. 11). The interactive interface (Fig. 12) is available for use with tools that do not have the sophistication to perform a batch transaction.

The tool/user interface needs to generate a transaction containing sufficient information for the data manager to perform both database retrievals and updates. The transaction contains two types of control entries: common and type specific. Common entries are those which are used in all data manager requests. The specific entries are determined

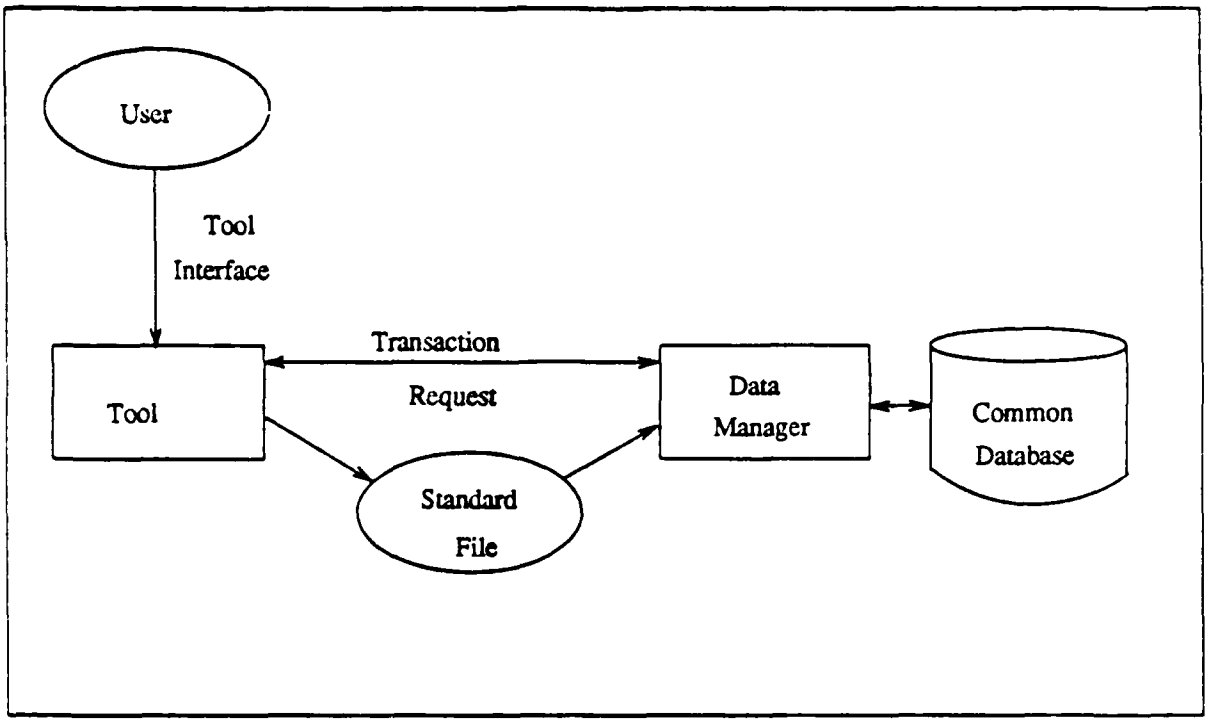


Figure 11. Batch Data Manager Interface

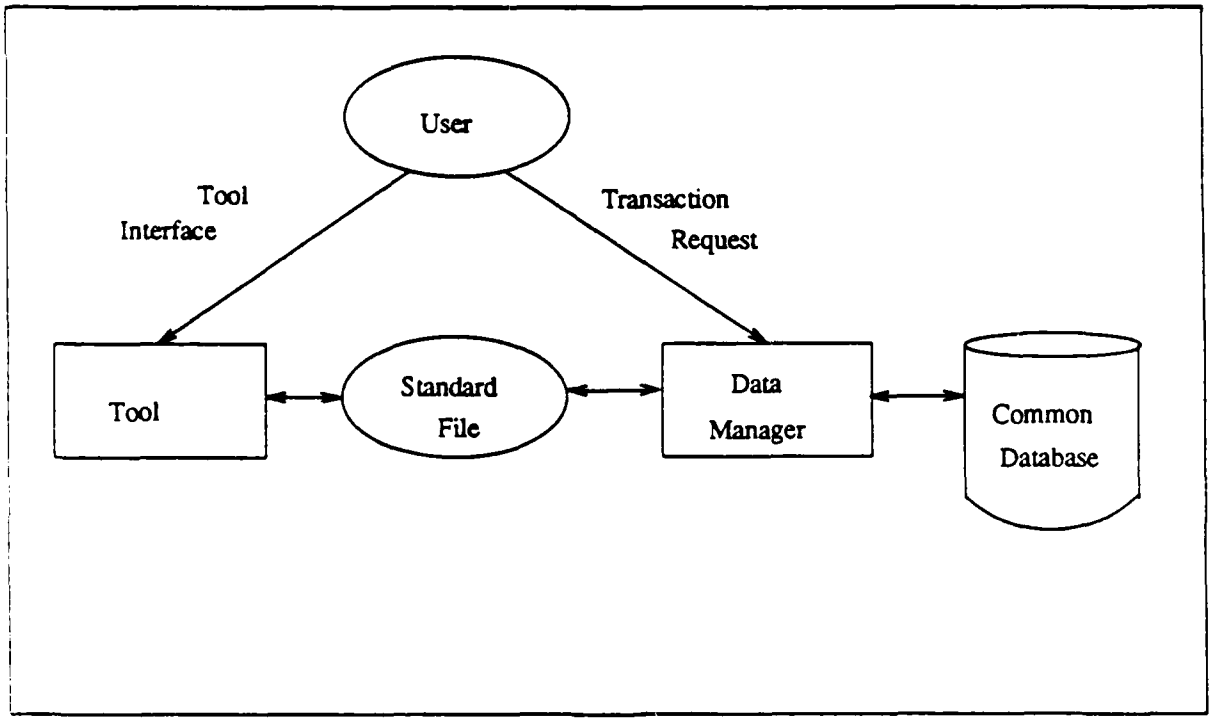


Figure 12. Interactive Data Manager Interface

by the type of request. The entries contained in a transaction request are discussed in terms of being in a file with the understanding that an interactive menu could provide the same type of information.

The common entries are those which identify the requesting tool, database name, standard data file name, type of transaction, requesting user, phase and type of data, and project name. The entries for an update request correspond to the common entries. The retrieval request requires additional entries to identify the data entity(s) to be retrieved.

The tool requesting a database transaction must be identified. As part of this identification, the phase and type of data is also needed. The phase entry is necessary to support multi-phase tools, such as the Data Dictionary editor. The type entry is needed to identify whether the tool uses action, object, or both types of data entities within a phase. This requirement is necessary because of tools which can use one or more types of data within a session (ie. activity and data item).

The database name is required to support the existence of multiple databases. This would allow for the databases to be separated for use on different computers or to isolate the data of a particular user or group. The only requirement for separating the databases is that each database contain both action and object entities within a phase. If

the databases are on separate systems, the user will be responsible for directing the transfer of transaction requests and tool files to the appropriate system.

The transaction indicator is needed by the data manager to control its actions. The basic types of transactions are retrievals and updates. A retrieval transaction will generate a standard data file and an update transaction uses the standard data file to perform database updates. Because the standard data file is used with both transaction types, the file name must be included in the transaction.

The identification of the requesting user is required for data access control. For retrieval requests, only the owner of a data item is allowed to retrieve it for modification, but all users may read the data. For database update requests, the user must be identified for the data manager to determine if the data can be updated by that user, and if not, prevent the user from modifying the data.

The final entry shared by both update and retrieval requests is the project name. Project name is part of the key for every relation in the database and must be provided.

The remaining entries are for use in retrieval request. These entries identify the data items to retrieved. The means to identify the required data items needs to be as flexible as possible. There are two ways to request the data. The most common is by explicitly listing all the data names. The second would be to identify a parent data entity

and return the data associated with this specific entity by levels of detail.

The ability to explicitly list the data entities needed is beneficial for tools which can only work with one entity at a time or with disjoint sets of data. However, the capability is also needed to list multiple entity names for those tools which can manipulate multiple entities within a session. As part of identifying the entity, the type (action or object) is required. This is necessary because of tools which simultaneously use both action and object entities.

The ability to retrieve subordinate data based on a high-level (parent) entity is required for tools such as the SADT editor. The SADT editor uses multiple action and object items during one session. It would be very difficult for the user to remember all the data entities associated with a session, but the user would know the parent name of the session's entities. Allowing the user to specify just the parent greatly eases the tool user's burden and reduces the possibility of errors created by including or omitting data entities.

An additional requirement exists for the capability to explicitly indicate the number of levels to be retrieved. This entry would be used in conjunction with the parent entry. This requirement exists for tools which can manipulate multiple levels of data.

The user interface requirements are that it allow both batch and interactive access and provide the following information to a tool:

1) Tool Identification
2) Database Name
3) Phase and Type of Data
4) Transaction Type
5) Standard Data File Name
6) User Identification
7) Project Name
8) Data Entity and Type
9) Parent Name
10) Number of Levels

The data manager needs to provide a tool/user the results of a transaction. The results need to reflect either the success or failure of the transaction and, in case of failure, identify the cause of the failure. Because the data manager supports both interactive and batch users, the results must be capable of being displayed to a screen during an interactive transaction or to a file during a batch transaction.

Data Retrieval. A primary requirement of the data manager is for it to retrieve data entities from a database for use by any tool using that database. The data manager must be capable of retrieving single or multiple data entities which are action and/or object entity types and,

most importantly, do so in a generic manner. Another important component of the data retrieval function is the support of session control.

To retrieve data entities from the database, certain key fields must be provided to the tool. These fields are project name, data name(s) or parent name and level, phase, and type. These entries uniquely identify the relations which are required to build a data dictionary entry. However, various tools may have different data requirements even within the same phase and type. To address this, some means must be provided to identify the necessary data elements to be written to the standard file, and the order in which these fields are to be stored. Whatever means is used, it must be generic so that common code can be used to retrieve the data for multiple tools rather than using tool specific code.

The means chosen to implement a generic retrieval must be flexible enough to handle any changes to current tools and the addition of new tools. This is an extremely important requirement because without it, modifying the data manager to incorporate tool changes could become too difficult.

As part of the generic retrieval method, the ability to retrieve multiple levels of information needs to be incorporated. The user should only be required to provide a key

data name identifying the top-level value from which the data entities are to be retrieved.

As part of the retrieval, a session control file needs to be established which tracks the data entities selected. This session control file must be assigned a unique identifier which insures no other tool could inadvertently destroy a session which it did not own. The session control file needs to identify the data entities used and their update status. The update status can be set to read or write if the entity's owner checks it out, otherwise only read permission is granted.

The session control file must also maintain other information. It must track what tool is using a particular session and the name of the session owner. This information identifies who has checked-out a particular data entity(s). This capability is necessary so that a user could be asked to check-in his session data for use by another tool or user.

Throughout the retrieval process, errors may occur. Any errors are to be reported to the user with a brief problem description. If a data item can be identified as the problem, its name is included in the error message. If a data entity in a multiple entity retrieval causes an error, it will be identified and the other entities are transferred.

A capability which may be needed by future tools is the ability to retrieve different versions of the same data entity. Currently, the database does not support the storage of multiple versions. Adding this capability is beyond the scope of this thesis, but the retrieval methodology needs to be developed to support this requirement.

Database Update. The requirements for the data manager's database update function are very similar to those of the data retrieval function. It, too, must provide a generic means to update the database, validate update request, and insure database consistency through session control.

To support the generic data description of a standard data file, a common representation, describing both update and retrieval formats, needs to be developed. This common data description would reduce the overhead of having separate files for updates and retrievals and prevent the problem of making a change in one file and not the other. These files should be developed so that the update code, like the retrieval code, can be common for all tools.

A requirement of the update function is to validate all data files submitted for updating the database. There are two types of updates received. One type is where the file has just been created by the tool with no existing session control file. The other type is the normal update where a session control file is available.

An update created by the tool should contain only new data entities. The entities are checked for completeness and use of unique data names. Three fields are required for any new data entity entry: project, name, and author. The project name is supplied by the user interface file. Name must be supplied by the user because it is part of the key for every relation in the database. Author is required to show entity ownership because only owners are allowed to modify an entity.

A regular update request uses the standard data file and the session control file created during the retrieval transaction. The session control file contains the status of each entity and validates whether the user owns the records he wishes to add, delete, or modify. If a tool cannot show an entity's update status, the session control file is used to direct the database updates. New entities are added. All entities in a write status are updated. Entities which were not included in the return file but were sent to the tool are retained in the database. Only explicitly identified entities are deleted.

The manipulation of data by various tools and the transmission of data files by data lines introduce the possibility for corrupted or invalid data to be submitted for update. The data manager must be able to detect and recover from such an occurrence. This requirement is limited to invalid data. It does not require the data

manager to perform consistency checking between entities. This is beyond the scope of this thesis and should be implemented via an external utility.

If an error is detected in an update transaction, the user must be informed as to what caused the error. If any entity within a session cannot be written to the database, the update is aborted and the database is restored to the state it was in before the update began. This is necessary to reduce the introduction of data inconsistencies into the database.

Common Database

There are only two requirements for the common database. The first requirement is for the relations developed by Thomas (16) to be modified to reflect the additional refinements made by Foley (4). The second requirement is for a means to indicate the read/write status of individual tuples within key relations. This requirement provides support of the session control file and adds additional information about the tuples for any non-data manager transactions.

Summary

This chapter has identified the requirements for integrating System 690 tools. The primary components are a standard data file and the data manager. The main requirements to be met are for the system to require minimal user

input, provide basic data protection, and be adaptable to change.

Adaptability is the key requirement. The standard file must be able to pass data to any tool in System 690. The format used must support new tools and changes to old ones. The data manager's retrieval and update functions must provide the same level of adaptability, but do so in a generic manner. As new tools are added and tool changes occur, the data description method and developed code must adapt without significant effort. Adaptability will be the driving requirement in the following design and implementation efforts.

IV. System Design

Introduction

The purpose of this chapter is to establish the high-level design based on the common database interface requirements. Chapter III identified the required components and functions which needed to be developed to provide the database interface. The system design addresses the overall structure of the interface and how its components interact. The design selected is not the only one available but it does meet all requirements. As part of the design, the system's transaction processing methodology must also be established.

System Structure

The system structure is based upon the components identified in the requirements. These components are the data manager, standard data file, and common database. This section examines each of these components based on their function in the system and on their inter-relationships to produce the overall structure shown in Figure 13.

Data Manager. The data manager's requirements are extensive. It must provide a generic means to perform data retrievals from and updates to a common database using the standard data file. It must support both batch and interactive tool transaction requests. Additional functions the data manager must perform are session control and error

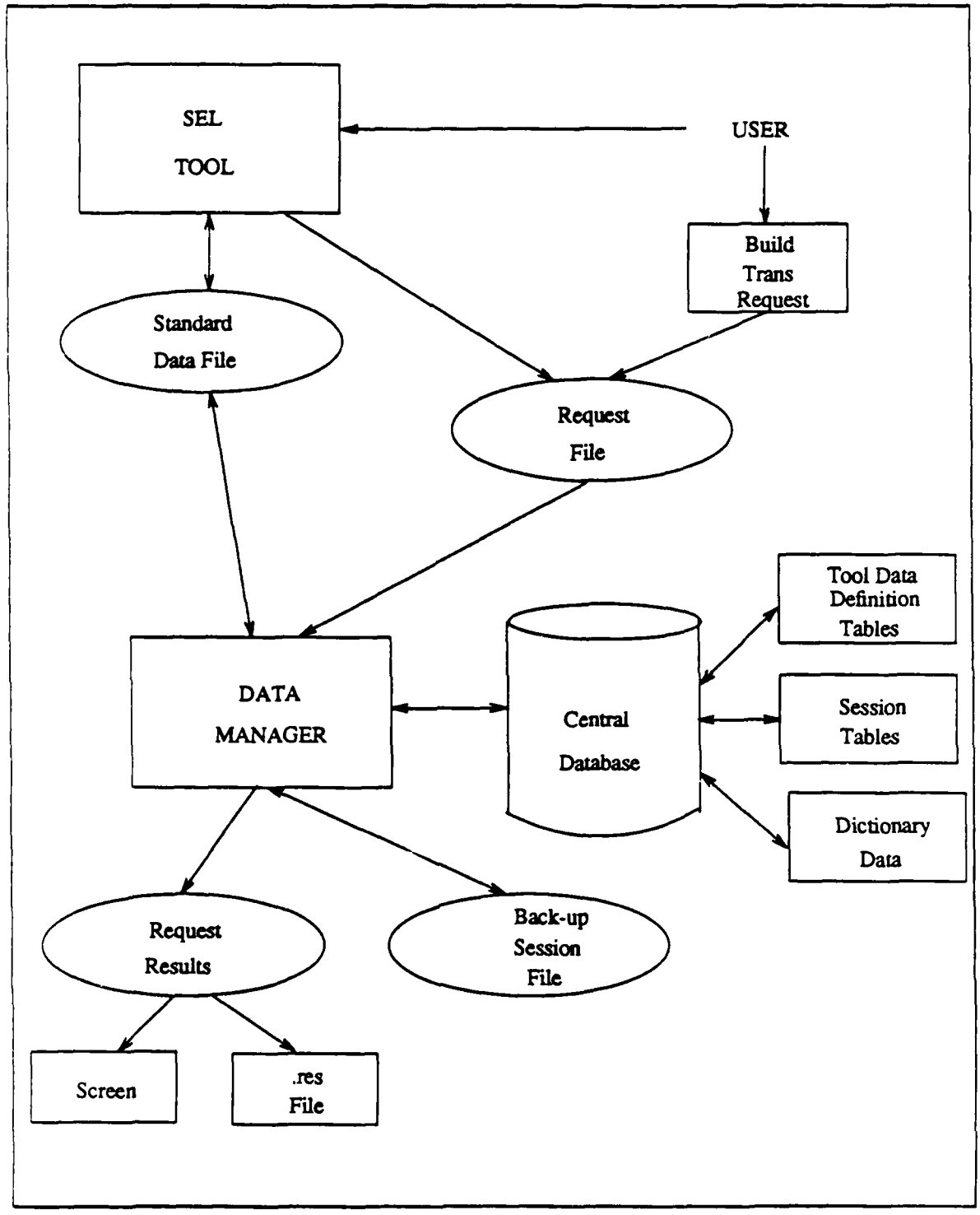


Figure 13. Overall System Structure

recovery. These functions establish the bulk of the system's structure.

The design of the generic database access is the key to the success of the data manager meeting the adaptability requirements. The method selected to support this requirement was a tool data definition table (Ingres relation). The definition table contains sufficient information to retrieve the data from the database, build the standard data file, and perform database updates using the standard data file. The data definition table can be adapted to support any tool and can be easily modified. Additional tables are used to describe the various tools and any unique processing needs a tool may have.

The tool-data manager interface is required to support both interactive and batch transaction requests. The method selected to support this requirement is the use of a Request File which contains the appropriate data manager instruction parameters. In the ideal case, this file would be generated by the tool without the user having to interact with the data manager, but this is not always possible. The current System 690 tools cannot support this, hence the need for an interactive interface as well. This interface will build the Request File interactively, so the data manager only has to support the file interface. An important part of this interface is the reporting of the results of any transaction request. The data manager provides the results to batch

transactions via a request Results File. Interactive transaction results would be displayed to the screen.

The session control function is dictated by the requirement to control access to the data. The control function occurs in two parts. At its basic level, session control must prevent the inadvertent modification of any data entities by any user other than its owner. The second part of session control is identifying and tracking data entities which have been "checked-out" of the database for modification and supporting the data manager's update function when the entities are "checked-in" to the database. The use of Session Control tables (Ingres relations) was selected to support this function. As part of the session control function, a back-up copy of the generated standard file is generated. This file is to be used by the error recovery routines to restore the database in case of severe errors when the data is checked-in.

The final required function is error recovery. The data manager maintains simple database consistency by not allowing incomplete or incorrect updates. In the case of an error, the data manager is required to restore the database back to its state prior to the transaction. Error recovery during a retrieval would be minimal because the contents of the database have not been altered, but update errors require more extensive procedures. To recover from such an update error, the back-up session file is used to restore

the database because it contains the structure of all the affected entities prior to the update.

Standard Data File. The standard data file is the data interface between the tools and the data manager. Its contents are dictated by the requirements. Every standard data file contains a file description header and the actual data entities. Note, however, that only the structure of the file remains constant. The actual contents of the data elements and their order varies from tool-to-tool and phase-to-phase. The data manager maintains the element's order and contents using the tool data definition tables.

Common Database. The common database is not affected in the design process except to support the addition of the data manager tool data definition tables and session control tables.

Transaction Processing

The system provides a common database interface. As such, the transactions that are processed will involve either the retrieval of or the update of data. Within the two transactions, there are four basic types: retrieval only, new write, retrieval for update, and write with update. The first two types are performed individually, but the last two are combined to form a session. This section examines each transaction type and their design considerations.

Retrieval. A retrieval is a read only action where any user may retrieve a data entity. The entity is not included in any session control relations because the entity may not be updated via a retrieve only transaction.

New Write. A new write is the result of a tool generating one or more new data entities to be written to the database. These new entities are in a standard data file which the data manager uses to perform the updates. Upon receiving a new write request, the data manager will need to check that none of the entities in the file currently exist. This is to prevent the accidental corruption of existing data.

Session. A session is the most comprehensive transaction the data manager must support. Each session consists of three steps: retrieval for update, tool manipulation of the data, and write with update. The retrieval for update retrieves the requested data entities into a standard data file (session file) which the tool uses to manipulate and modify the data entities. After completing its modifications, the tool submits the standard data file to the data manager for writing. The data manager monitors each session using the session control files to track the entities which were checked-out and the user's name. When the session file is resubmitted for write update, the data manager needs to check for any invalid updates. If none occur, the session is terminated and the data entities are checked-in. If an

error is encountered, any updates made by the session file are removed and the session back-up file would be used to recover the database.

Summary

The purpose of the system design was to provide the basis for the detailed design specifications for a common database interface. As shown in Figure 13, the key components are the tools, data manager, standard data file, and central database.

The importance of the detailed design is to not just show the components, but to identify how they interface. Also shown are the work files and relations used by the data manager in supporting the generic database interface, session control, tool interface, and error recovery. The detailed design of these components, files, and relations are provided in Chapter V.

V. Detailed Design

Introduction

The purpose of this chapter is to provide a detailed design of the components specified in the system design chapter. The major components identified were the standard data file, data manager, and common database. Within the data manager requirements, four key sub-components were identified. The first was a generic tool data definition table. The second data manager component was the tool/user interface. The third was a means to support session control. The last component was a means to recover from errors. The design of each of these components and sub-components is presented in this chapter.

Standard Data File

The standard data file serves as the primary interface between System 690 tools and the data manager. The requirements of this file indicated that it should consist of a file description header and a set of formatted data entries containing the required data elements. This section examines the design of these two components and their overall structure.

File Description Header. The required contents of the file description header were identified as the following: session identification, tool/file compatibility header, project, phase, type, data entity summary, and start/stop

time entries. These required fields were combined to produce the file description header (Fig. 14) used in the standard data file.

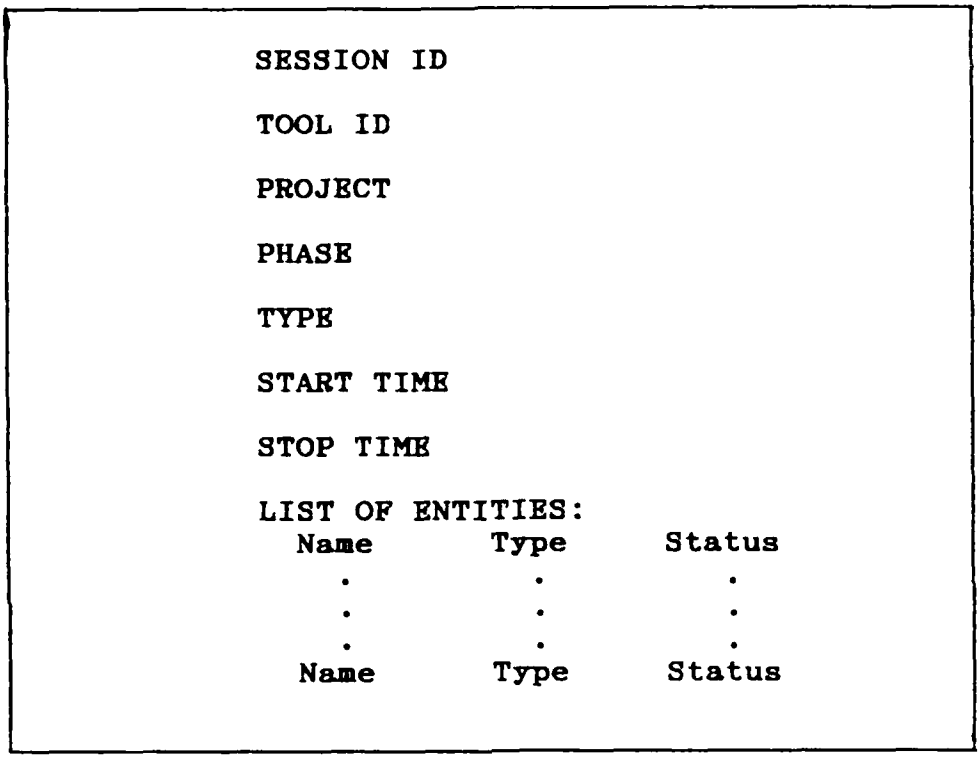


Figure 14. File Description Header Format

The requirement that the standard file be used by both tools and the data manager places certain demands on the design and use of the header fields. The general issues are addressed below with the detailed field formats being discussed in Appendix B.

Session Identification. The session identification contains either the session identifier assigned by the data manager at the beginning of a session (retrieval for

update) or a standard entry indicating the file was built by the tool. The identifier assigned by the data manager is used to control database updates and must be unique for each session file to prevent using the wrong session control information in controlling a write with update request. Because of this, the tool must maintain this identifier for later submittal for database updates. The standard entry is used when the data file has been created by the tool to enter new data into the database. The standard data entry is constant for all tools and the data manager will expect this entry for new writes. Without a valid session identifier, assigned or standard, the data file will be rejected.

Tool Identification. The tool identification code is needed by both the SEL tools and the data manager. A tool can use the tool code to verify the file contains data properly formatted for its use. The data manager uses the tool code to help determine the format of the data entities within the standard data file.

Phase Indicator. The phase indicator will contain only one phase. This method was selected even though some tools may require data from more than one phase. The single phase per data file allows the different phases to be contained in separate databases residing on different systems. The data manager cannot manipulate data across systems in a single session. For this reason, a separate data request must be made for each phase.

Type Indicator. The type indicator designates whether the data entries in the file are action, object, or both types of data entity. Since all can be handled, this is needed only to allow more efficient operation of the data manager.

Start/Stop Times. These times are initialized by the data manager but the tool provides the actual values. The start and stop times represent the total time used by a tool during one session. The method used to generate the entries are tool specific to allow for the most accurate representation of a tool's specific usage patterns.

Data Entity Summary. The data entity summary can consist of multiple entries. Each entry contains the name, type, and status of each data entity contained in the standard data file. The entities are ordered by type with all action entities occurring first, followed by the object entities. This grouping was selected to ease file handling by presenting a consistent ordering.

The status entry indicates the status of each entity in the standard data file. The acceptable statuses are read, write, and delete. The status of an entity is determined by its intended use and whether it is in a standard data file generated by a tool or the data manager.

A read status occurs only in standard data files generated by the data manager. This status is used when a requested entity cannot be updated in this session.

A write status can occur in either a tool-generated standard data file or in one generated by the data manager. All entities in a tool-generated file are in a write status because all the file entities are supposed to be new. There is no need for any other status in a new write. In a standard data file generated by the data manager, the write status indicates to the tool that the data entity can be modified.

The delete status occurs only standard data files generated by the data manager and modified by a tool. The delete status allows a tool to delete entities during a session.

Because some tools may not have the sophistication to indicate the status of an entity, the data manager assumes, on new updates, that every entity in the file is submitted in a write status. If the file was checked-out, the appropriate session control files are used to control the updates when the standard data file is checked-in.

Data File Entries. The data portion of the standard data file consists of one or more data entity entries. Each entry is composed of all the data elements necessary to satisfy a data dictionary entry. The data elements are contained in a series of data records (Fig. 15) and consist of the fields identified in the requirements. The file contains the data elements for all entities identified in the file description header entity list, except for those in

a delete status. The entities marked for deletion do not have a corresponding set of data records in the standard data file. The general design issues for the data entries and the overall entity structure are discussed, while the detailed field formats are addressed in Appendix B.

Data Name. The data name corresponds to the data element's attribute name. A tool returning this record must have this name correct or the data manager will reject the entity.

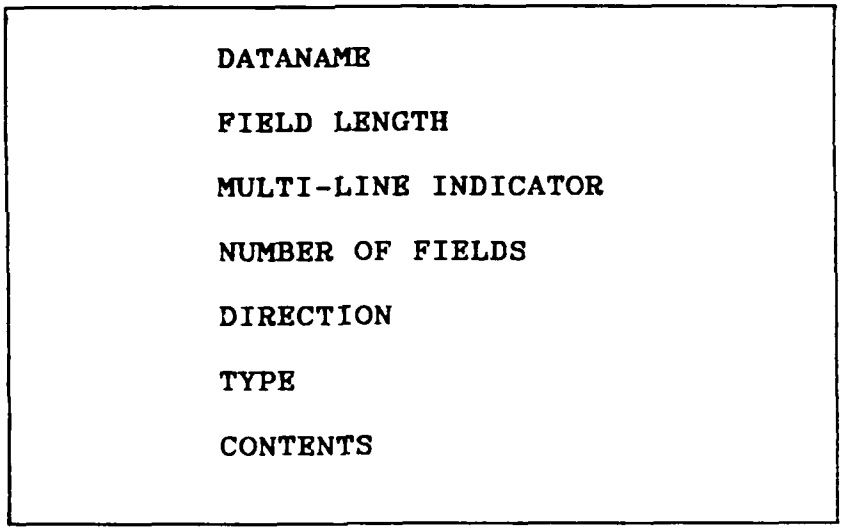


Figure 15. Data Element Record Format

Field Length. The field length entry contains the data element's maximum Ingres field length. The requirements indicated the need for this to inform a tool the content's length to allow tools with varying field length support to manipulate the entries.

Multi-line Indicator. This field contains either a Y or N to indicate that the data element is part of a multi-line field.

Number of Fields Indicator. This field contains the sequential identification number of single-line fields occurring within a group field. The numbers are in descending order to allow a tool to know how many fields to expect.

Direction and Type Indicators. These indicators correspond to the direction and type attributes of certain data elements. These fields are required to form the data element's key. Because of this, the tool submitting the file must provide the proper values in these fields. In entries where these fields are not needed for database access, the field, while present, is ignored.

Data Contents. The data contents field contains the data element value. The field may contain up to 60 characters. This limit was selected because of the Data Dictionary Editor. The editor is the most heavily used tool and cannot manipulate fields longer than 60 characters (16: 57).

Entity Structure. The data element records within an entity have a specific order. The first element record in an entity is the entity's name. The name occurring first was selected because it is standard for all data dictionary entries and it provides a quick means to identify the entity name of the data elements. The order of the remaining data

elements is dictated by the tool's requirements. The ordering is controlled by the tool data definition table.

The order of the data element records must be maintained upon submittal for updates. The data manager expects the data elements to be present and in a specified order, including empty contents fields. If a data element is not present the file is rejected. The purpose of this is to prevent posting any incomplete entities to the database.

Data File Structure. The standard data file structure (Fig. 16) is built using the file description header and data entity entries. The file contains all ASCII characters and consists of the file description header, data entities, and section delimiters. The delimiters are unique for each section and are designed to help the tools and data manager maintain their position in the file. The delimiters also help tool developers read the file's contents for debugging purposes.

Data Manager

The data manager is the key to providing an integrated environment within the Software Engineering Laboratory. The primary components of the data manager are the tool data definition table and the tool description table. These tables permit the generic classification of the entities used by the tool. These tables are used to support the data manager's two primary functions which are to perform the database retrievals necessary to generate the standard data

```
#@@BEGIN@@#
#@#HEADER BEGIN#@#

<file description header, Fig. 14>

#@#HEADER END#@#
###ACTION TYPE###

@##START##@

<entity element record, Fig. 15>

@##STOP##@

    o
    o
    o

###ACTION END###

###OBJECT TYPE###

@##START##@

<entity element record, Fig. 15>

@##STOP##@

    o
    o
    o

###OBJECT END###

#@@END@@#
```

Figure 16. Standard Data File Format

file and to use the standard data file to perform database updates. A tool/user interface is also required to control the actions of the data manager. The design of the tool data definition table and the tool description table are

examined first. The three data manager functions are then discussed. For additional details concerning the data manager and its interface, reference the User Manual in Appendix C.

Tool Data Definition Table. The data manager requirements identified the need for a means to support the retrieval of data dictionary entries, formatting the retrieved data into the standard file format, and updating the database. The method selected was also required to be flexible in design to incorporate current and future tool data requirements with little or no programming. These requirements provided the basis for the following design of the tool data definition table. For detailed field formats and values, refer to Appendix C.

Table Usage. A tool data definition table contains the information necessary to describe a single data entity type to the data manager. The table is tool, phase, and type specific. Therefore, a tool using both action and object entities within a phase requires two data definition tables, each with unique relation names, to describe its data entity formats.

Table Format. The primary issue in designing the data definition table (Fig. 17) was supporting the data manager's database transactions. The table had to provide sufficient information to access a data element. The minimum information required is the following: data element

name, element's relation name, relation's key names, and the entry classification of the element's relation. Each of these items require one or more entries in the table and are discussed below.

The data name, relation, and key fields each contain the appropriate Ingres value needed to access a single data element. Data name also corresponds to the data name entry in the standard file data record. Relation is the name of the relation containing the data element. The key fields

DATANAME	RELATION	KEYFIELD_1	KEYFIELD_2

FIELD_DESCRIPTION	ENTRY_CLASS	MULTI_LINE_INDICATOR

NUMBER_OF_FIELDS	DIRECTION	TYPE

DELETE_FLAG	VERSION	LINE

Figure 17. Tool Data Definition Table

are the attribute names used in a relation as keys. For all current tools, only the first key field is used, but a second key field was provided to support future tools which may require it.

The data definition table entries just identified supply only a portion of the information required for data retrieval and update. Information describing the data's relation structure is needed to perform accurate and efficient database transactions. The entry class field performs this function. By classifying a data element's relation, the data manager can perform its database accesses according to class, eliminating special coding for each data element. The use of entry classes is the key feature of the data definition relation. The relations used in the various data dictionary entries fall into general classes. By classifying a relation by its structure and not specific code values, the amount of code required to read and write information is greatly reduced. Furthermore, new relations can be added to the database by either creating a new class or by using an existing class. This flexibility provides the generic capabilities identified in the requirements.

The direction and type fields are used to aid the data manager in accessing data elements whose usage is determined by its direction and/or type attributes. The paname data element, contained in the processio relation (reference Appendix A), is an example of such an element. The direction and type field values are used for database transactions and in the direction and type fields contained in the standard file data elements.

The delete flag indicates which relation names and key fields to use in deleting the elements associated with a data entity. Only a limited number of the table entries have to be marked for deletion because several table entries may correspond to a single relation (ie. ALIASNAME, COMMENT, and WHEREUSED in the paalias relation). This grouping uses the same relation and key field names for a deletion. Using the Ingres delete command will remove all associated tuples. Another reason this method was selected is certain entries in the data definition table are not a member of the entity's type class but are needed to provide a complete data dictionary entry. An example of this are the SOURCES and DESTINATIONS entries in a data item data dictionary entry. These fields are part of the activity entity and cannot be deleted by a data item transaction. By marking these entities as non-delete entries, the elements may be used in both types of dictionary entry.

The version field is used in accessing data elements which have multiple versions. The entry contains the attribute name used in a tuple to identify the version of its contents.

The number of fields entry indicates how many elements are to be retrieved from a single tuple. This entry allows the data manager to access all the data elements in a relation with only one transaction versus one for each data

element. The number of fields is also used in the standard file data record.

The field description and multi-line indicator are provided to support the generation of the standard file data records. Certain tools (i.e. Data Dictionary Editor) require these indicators to structure the display format of the element.

Element Entry Order. The order in which the various data elements are written to the standard data file is based on their order of entry within the tool data definition table. The data definition table allows the elements in a relation to be split in the standard data file. To support this, the data manager will have to perform multiple retrievals of the same relation to get all the required data elements. This introduces inefficiencies into the data manager but eliminates the requirement of a tool needing to know the structure of the database.

Tool Description Table. The tool description table (Fig. 18) describes a tool and its data needs to the data manager. The description table is used by the data manager for transaction request verification and database retrievals and updates. There is a tool description table entry identifying the tool data definition table for each phase and type of data entity used by a tool. This is required because a data definition table only describes a single data dictionary entry.

TOOL_NAME	PHASE	TYPE	DEFINITION_TABLE	DESCRIPTION

Figure 18. Tool Description Table

The tool name contains a code which uniquely identifies a tool. The same code will be used for multi-phase tools to prevent having the tool submit a different code for each phase it uses. The tool name, phase, and type fields are used to identify the specific data used by the tool. The definition table field provides the data manager the Ingres relation name identifying the appropriate data definition table(s) to use in retrieving a tool's data entities. The description field provides a means to better identify a tool and its use and is for documentation purposes only.

Tool/User Interface. The tool/user interface provides the means for tools or users to perform database transactions. The design of the interface addresses the format of a tool data request, the types of interface options to be made available, and error reporting procedures.

Tool Data Request. The requirements for the tool data request contents were established in Chapter Three. These requirements helped determine the tool data request format (Fig. 19). The entry requiring further clarification is the transaction indicator.

The transaction indicator informs the data manager of the types of actions it is to take. The system design

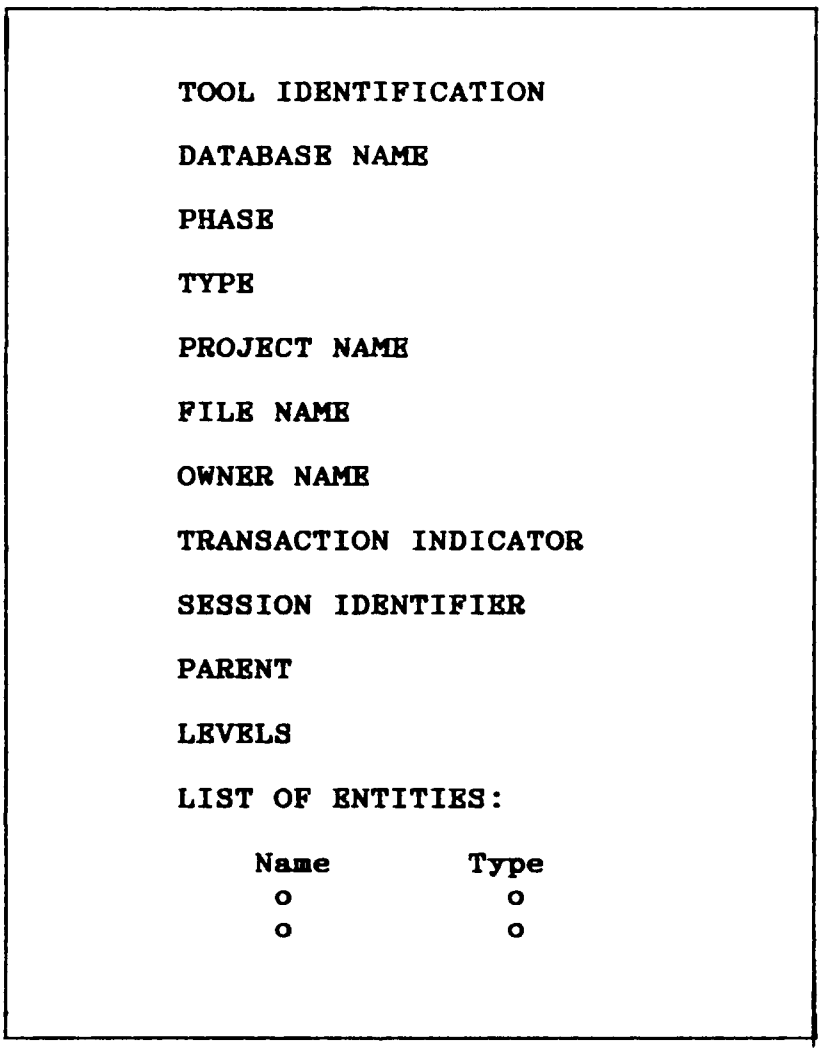


Figure 19. Tool Data Request Format

identified four types of transactions: retrieve only, new write, retrieve for update, and write with update. These four transactions perform all the required transactions but two other transactions were identified which would improve the data manager's "user-friendliness".

The first transaction added is the delete function. This allows a tool/user to provide the data manager a list

of entities to be deleted without having to retrieve them for update, changing their status to delete, and resubmitting them for write with update.

The second transaction added is a session abort function. This was added to allow a user to abort a session without submitting the session file. This transaction was added for two reasons. The first was to support easy database maintenance by providing the database administrator a means to delete old sessions. The second reason was to allow a user to delete a session in case the session file is corrupted or lost. As part of this transaction, the session identifier is required to identify the appropriate session.

Interface Design. The data manager is required to provide a user the option of using either an interactive or batch interface. The interactive interface will provide the user a series of menus to build a transaction request file (Fig. 19). The batch interface is provided to support tool generated transaction requests. The batch request and the interactive request files both have the same format.

A design decision was made to implement the interactive interface in a separate program. This design allows the data manager to process interactive and batch transactions the same way. This common interface simplifies the data manager design by eliminating interaction with the user during transaction processing.

Results Reporting. All transaction results are reported through the use of a results file. The results file name consists of the transaction request file name with a .res extension. The results file will contain the list of successfully performed transactions. In the case of an error, the cause and error recovery results are placed in the results file. The exception to this is during interactive processing when the results are displayed directly to the screen.

Data Manager Retrieval Function. The data manager performs all the data retrievals required by a tool. To perform these retrievals, the data manager must validate the request, provide session control, identify the data entities to be retrieved, retrieve the data, and generate the standard file. Each of these functions is an important design concern.

Request Validation. Request validation occurs in two steps. The first step is checking the validity of the transaction request (Fig. 19). The second step is determining if the requested data entities exist.

The required transaction request entries are dependent on the type of transaction being performed. The entries required for any transaction are the tool identification, database name, and owner name. The other entries are transaction dependent.

The retrieval entries are the session file name, project name, phase and type of data, and the entities to be retrieved. The write transaction has the same required fields but does not list the entities because the entity list is contained in the standard data file header. A delete transaction uses the same entries as a retrieval request. A session abort transaction only requires the session identifier.

The transaction entry validation checks are limited. The tool identification, phase, and type are checked against the tool description table for accuracy. The other fields are checked for their presence. Any errors encountered in the validation generate an error message and cause the data manager to terminate. If no errors are encountered, the existence of data entities identified for retrieval or deletion is checked. For other transactions, the data manager by-passes the existence check and begins processing.

The data existence verification insures at least one of the requested data entities exists. The transaction may either identify the data using the parent and level fields or the data entity list. These are mutually-exclusive entries. If a multi-level retrieval is requested, both the parent and level fields must be present. If the multi-level fields are not used, the data entity list must contain at least one entry. If these conditions are not met, the

request is rejected. Otherwise, the presence of the requested entities is checked.

The data existence verification insures at least one of the requested data entities exists. A multi-level retrieval uses the multi-level transaction table (Fig. 20). The tool name, phase, and type fields correspond to the transaction request entries. There are three types of multi-level request. The first request type is for a retrieval of both action and object entities. The second type is for a retrieval of only action items. The third type of request is for object entities only. These are hierarchical retrievals based on the number of levels. A tool may request no more than the number of levels allowed in the multi-level transaction table.

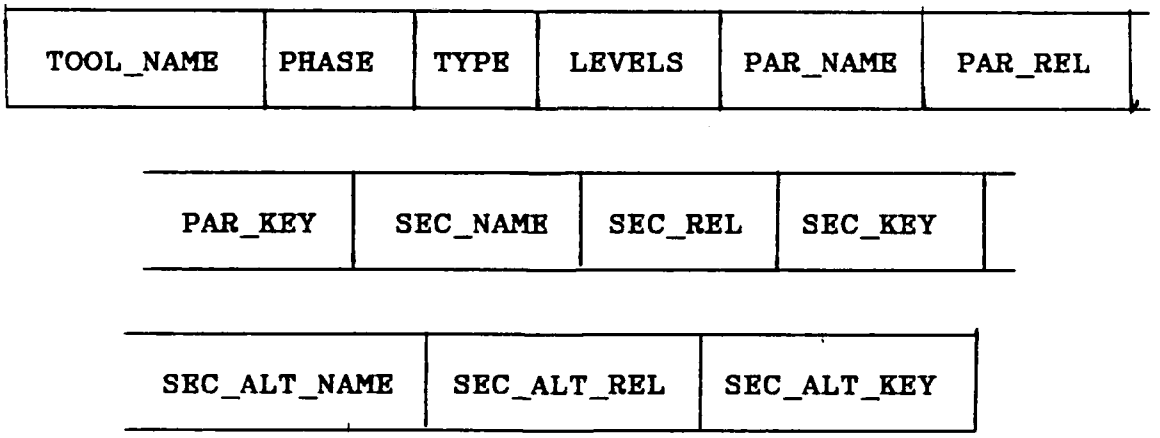


Figure 20. Multi-Level Transaction Table

The multi-level transaction table supports a single entity type (action or object) retrieval via the parent relation information. The secondary relation information is

used for retrieval of both entity types. The secondary information is used to retrieve the objects pointed to by the action parent or vice-versa. This mechanism supports the bulk of the retrieval request but the situation can arise where an action entity points to objects contained in more than one relation. To support this, an alternate relation is provided. An example of this type of use arises in the design phase. The objects pointed to by a parent process are contained both in the processio relation and the papassed relation. In this instance, all the table's relations are used to retrieve the information.

A key design decision was made not to support the retrieval of an entity's aliases. Alias retrieval is not supported for two reasons. First, the use of aliases is a poor software engineering principle and is provided in the data dictionary only to support the occasional problem which arises when two large systems, both using a similar function but with different names, are combined and the effort to change one system's references to the entity is too extensive to be warranted. Second, both the data manager and the tools would require a much higher degree of sophistication to resolve aliases. The effort to develop this sophistication versus the benefits derived dictate that alias retrieval not be supported.

To show how a multi-level retrieval is performed, a sample retrieval of an activity and its data items is

presented. Before examining the sample, note it is only for the requirements phase. Different relations would be used for the other two phases.

SAMPLE:

ahierarchy	activityio
project	project
hianame	diname
loaname	aname

Parent Name	: loaname	Secondary Name	: diname
Parent Relation	: ahierarchy	Secondary Relation	: activityio
Parent Key	: hianame	Secondary Key	: aname

The transaction parent value is used as a key (hianame = "parent value) to identify all the parent's subordinate activity names (loaname). These names are placed into a list. If the number of levels is two or more, these names are then used to retrieve the next level of subordinate activity names. This process is followed until the requested number of levels have been retrieved or there are no more subordinate activities.

The list of activity names is used as a key (aname = "activity name") to retrieve the associated data items (diname). After the activity name list is exhausted, the data manager uses the activity and data item names to perform its retrieval functions to build the standard data file.

*NOTE: The above example was for a retrieval request. The multi-level transaction table can also be used in a delete transaction. This provides a means to easily delete an entire level of dictionary entries from the database without having to explicitly identify each activity and data item.

The sample showed a typical multi-level retrieval for both entity types. A key aspect of the retrieval is the selection of the parent and secondary relations. In an object only retrieval, the dihierarchy relation would be the parent and not activityio.

Session Control. Session control is an important part of the data manager retrieval function. The retrieval portion of the data manager determines the status of all requested data and generates the session control information. This session information is maintained in two tables: session entity table and session identification table. This section examines the two session control tables and the data manager's use of them.

Session Entity Table. The session entity table (Fig. 21) is designed to track each entity used in a session, its type, and update status. The session id corresponds to the associated session identifier.

SESSION_ID	NAME	TYPE	STATUS

Figure 21. Session Entity Table

The name and type fields reflect the data entity's name and whether it is an action or object entity. The status reflects the update status of the entity. If the session owner also owns the entity and the entity is not checked out, the entity is placed in a write status, otherwise it is placed in a read status. Entity ownership is based on the entity's author name and the session owner contained in the tool transaction request.

Session Identification Table. The requirements identified the need to maintain the status of a session, describe the type of data used in a session, and identify the session's owner and tool being used. To satisfy these requirements, the session identification table (Fig. 22) was designed.

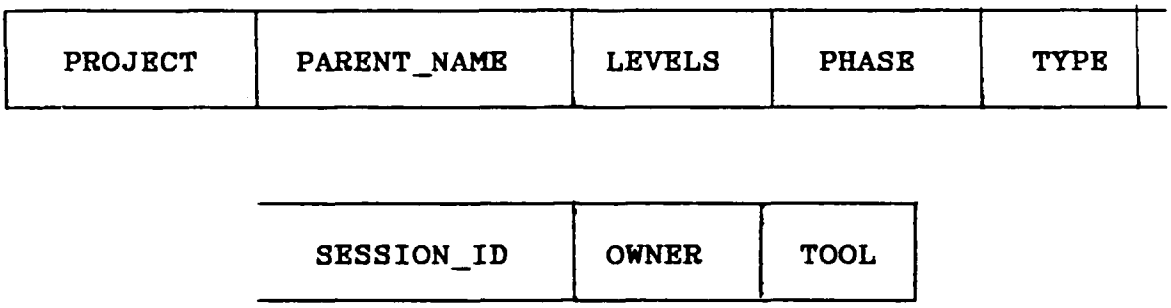


Figure 22. Session Identification Table

The project field contains the project name, which when combined with the entity name, can access all the data elements in the entity. The phase and type fields identify the data dictionary entry being used.

The parent name and level fields are for tools which require multiple levels of data to be retrieved. An example of this would be to retrieve all the activities and data items associated with an SADT chart. The parent name field contains the data value used in the retrieval. The level field indicates how many levels of data were retrieved below the parent level.

The session identifier must be unique for every session. Therefore, a date/time stamp is used to designate a session. The format of the field is aMMDDYYHHMMS (i.e. a09198708125). This format allows the session identifier to be used as an Ingres relation name, providing a means for future tools to create session specific relations based on the session identifier without having to convert the identifier to an Ingres acceptable form.

The owner and tool fields correspond to those in the transaction request and provide an easy means to identify the session's owner and tool. This satisfies the requirement that checked-out data may be easily located. This should facilitate group efforts by allowing the team members to find needed information and coordinate with its owner to check the data back in. These fields are also used by the data manager in validating update requests.

During a retrieval transaction, the data manager is responsible for updating the session identification table and the session entity table. The session identification table entries are filled in using information contained in the tool data request and tool description table. The only field not provided is the session identifier. The data manager will assign this value based on the current date and time.

Data Identification. The data to be retrieved is identified during request validation. The request valida-

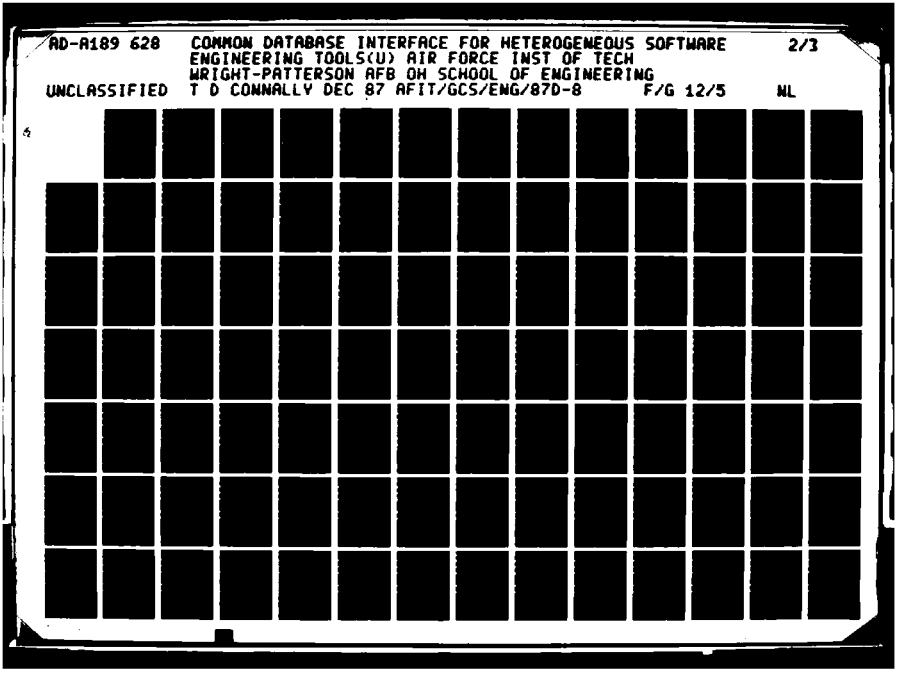
tion generates a linked list containing all the valid entities. This list is used to retrieve all the data.

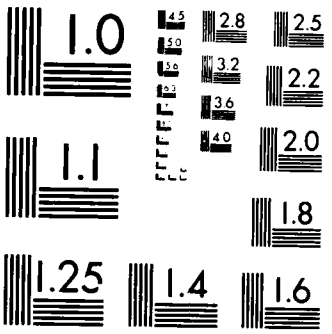
Data Retrieval. The retrieval of each data entity is controlled by the appropriate tool data definition table. The retrieval function is designed to perform all retrievals according to the entry class. As each entity is successfully retrieved, its status is updated in its status attribute. This attribute occurs in the entity type relation, i.e. activity or parameter.

If an error occurs, the data manager identifies the error and records the error and its cause in the results file. The status of the data entities which have been retrieved to this point will be restored and the session control tables corrected.

Standard Data File Build. The file description header is written to the file before any data retrievals are performed. Once the header is successfully written, the data entities are written to the file. The entities are written in the order they occur in the header with action entities occurring first.

To support error recovery in the update function, a copy of the standard data file is created at the successful completion of the retrieval function. The session identifier is used as the file name. The file is deleted whenever the session of the same name is removed from the session control tables.


AD-A189 628
COMMON DATABASE INTERFACE FOR HETEROGENEOUS SOFTWARE ENGINEERING TOOLS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING
2/3
UNCLASSIFIED
T D CONNALLY DEC 87 AFIT/GCS/ENG/87D-8
F/G 12/5
NL



Data Manager Update Function. The update function's requirements identified the major components needed to perform database updates. These components are request validation, database update, database housekeeping, and error recovery. A major concern in the design of these components is whether the update is using a standard data file that is part of an existing session or if it contains all new data. The two types of update and their effect are considered in the discussion of the four main update components.

Request Validation. Validating a tool update request consists of checking both the contents of the tool update request and the information contained in the standard data file. The validation includes checking for the presence of required fields, that the fields in the request and header match, and the requested transactions can be performed.

Every field in the update request is required, except for the parent, level, and data entity fields. These fields are optional and are only used for deletions. This allows the user to delete various entities without first building a session file.

The tool identification, phase, and type fields are checked against the tool description table for validity. The remaining fields are checked for their presence and their validity against established formats (reference

Appendix E). The transaction indicator is especially important because it controls the steps taken in the database updates.

Every field in the file description header must contain a valid entry. If the transaction contains all new data, the session identifier must also reflect this. Otherwise, the session identifier must match an existing session in the session identification table. The tool identification, phase, and type fields must also match the values in the session identification table. If the file description header values do not match those in the tool request, an error is generated and the update is rejected.

After the control fields pass the validation check, the entities listed in the file description header are checked to see if the indicated update can be performed. The requested update status of an entity is checked against the status maintained in the session entity table. If the two statuses are incompatible, the update transaction is rejected. When a new entity, either added in the session or part of a tool originated session, is to be written to the database, the database is checked to identify if an entity exists with the same name. If one does, an error is produced and the update is rejected.

Delete transactions require an extra step. The entity list must be generated from the parent and level entries or the data name entities contained in the tool request. This

list is used in place of the file description header's entity list. All identified entities must not be in a write status. If any entity is, it cannot be deleted by a non-session request. If any entity in the list cannot be deleted, the update is rejected.

Database Update. The data manager database update routines support all database transactions. There are three types of database updates made. These types are the addition of a new entity, the deletion of an entity, and the modification of an existing entity. The design of the update routines is based on supporting these transaction types.

All the update transactions are controlled by several tables and lists. A linked list is created which contains either the entity list contained in the standard file description header or the deletion list generated during request validation. The entities contained in the standard data file are expected to occur in the same order as the entity list. If an entity is listed for update in the entity list but is not in the data file, the update is rejected. Entities in a delete status do not have a corresponding data entity in the standard data file's entity list. The tool data definition table provides control of the actual updates of the various relations constituting a data entity. If a data element listed in the definition table does not occur in the data file, the file is rejected.

The addition of an entity is straightforward. The data elements contained in the data file are placed in a structure. If all the elements are present, the data definition table is used to control the database update. This method allows only correct entities to be written. If an error occurs building the structure, error recovery is simplified by not having to recover from a partial update.

The file deletion routine uses the tool definition table to identify the relations containing the data elements to be deleted. If an entity is identified for deletion, but is not in the database, no error is generated.

The file modification routine utilizes both the delete and add routines. The update process occurs in two phases. The first is the deletion of the old entity. The second phase is the addition of the modified entity. This method was selected over trying to modify only the affected data elements for simplicity and efficiency. To update only modified data elements, the tool would have to track the update status of each data element or it could identify the entity as containing modified elements. The current tools do not have the sophistication to maintain the status of each data element, so the data manager would have to identify the modified elements. The overhead involved in either method far exceeds any benefit which would be derived from reducing the number of relations updated.

After the database has been successfully updated, the various work files and session control tables must be updated. The work files are deleted, including the copy of the standard data file made during session generation. The session entity table entries for this session are also deleted. The session identification table is updated by removing the information associated with the session's identifier.

In the occurrence of errors, two types of actions can be taken. Errors identified during request validation only require the transaction error file be built and the update be terminated. If an error occurs during the updates, the database must be restored. The standard data file copy is used to recover any entities which were deleted or modified. The work file is also used to delete any new entities added to the database. The appropriate error messages are generated and the update terminated.

Common Database

The basic design of the common database is well explained in Thomas' thesis (16: 84-142). The data manager requires a few extensions to this design. These extensions are required to support the various tables used and to enable entity status tracking. (Reference Appendix A for the current entity relations and formats.)

Thomas intended for only one database to be used for all three data dictionary phases. However for flexibility,

this is no longer the case. A database may now contain one or more phases. The only requirement is that both entity types used within a phase be present.

The tool data definition table and the session control tables must also be supported by the database. These tables (relations) only have to contain sufficient information to describe the entity types stored in the database and support the tools accessing the database.

Summary

The data manager design focused on the data manager's functions, the format of the various tables used to track and control database transactions, and the common database. The goal of the design was to provide an easy to use system which provides the user the flexibility needed to support his various needs.

The tables are easily modified and changes have minimal impact on the data manager or database. The data manager can perform its various functions by using these tables, and when changes occur, significant code changes are not needed. The error recovery provided is not sophisticated but it maintains basic database consistency. The design impact on the common database was also minimal.

The design of the various components was oriented towards maintaining system simplicity. The implementation of the design and the results are discussed in the following chapter.

VI. Implementation, Test, and Evaluation

Introduction

The primary goal of this thesis was to develop a working data manager for use in the SEL. This chapter examines the implementation issues of the data manager, reviews the testing procedures used during development, and evaluates the data manager from the tool developer's viewpoint and its performance.

Implementation

The data manager implementation had to address several issues. The first of these issues was the development environment of the data manager. Another issue addressed was the implementation of the data manager interface. The final issue addressed was the error detection and recovery methodology used in the data manager.

Environment. An objective of the data manager was for it to be flexible and widely available to the SEL tools. The AFIT computer environment and this objective dictated the operating system, programming language, and DBMS to use in the data manager implementation. Within AFIT, the most widely available computer systems are VAX 11/785s using the Berkeley Unix 4.3BSD operating system. Common to these systems is the Ingres DBMS. This availability directed the selection of Unix and Ingres for use with the data manager. Based on this selection, the programming language had to be

C because it is the only language available which can directly interface with Ingres. The interface between C and Ingres is implemented using Embedded Quel (EQUEL), which can be written directly into the C programs.

This environment supported the data manager development very well, except for one weakness. The EQUEL pre-processor would not allow a data item to be declared using the same data name in separate functions within a single Unix file. EQUEL would recognize only the first data item declaration and indicated any subsequent declarations as multiple definitions of the data item. This problem mandated the use of a different data name in each function, even though the data item was used in the same manner in each of the functions.

Interface. The data manager interface implementation had to address two problems. The first problem was providing a user acceptable response time for interactive requests. The second problem was preventing a user from improperly terminating the data manager and consequently corrupting the database. The solution to these two problems had a major impact on the data manager implementation.

The problem of providing a user acceptable response time was a significant issue. Previous efforts (4, 16) showed that poor response time caused user dissatisfaction. The method selected to solve this problem was to develop an interactive menu program which generates a transaction

request file and keeps computer interaction to the minimum. The data manager then uses this transaction file for background processing. Background processing of the transaction frees the user from having to wait at the terminal for the transaction to finish. This is especially important for transactions processing a large number of entities which may require several minutes on a heavily used system.

The transaction file interface with the data manager permits the interactive generation of a request but also supports tools which can generate transaction requests without user interaction. These tool-generated transactions can be executed in batch mode, freeing the user from interacting with the data manager.

The ability of the data manager to run in the background also solves the improper program termination problem. Unix provides background job execution which allows a program to run without the user monitoring its execution. By executing the data manager in the background, the user cannot accidentally terminate its execution. A feature of background jobs in Unix is the program can continue processing even if the job initiator's connection is terminated. This is an especially important feature considering the number of jobs which will be executed by remote users via modem connections.

The data manager and Ingres provide the concurrency control necessary to support multiple simultaneous users. Ingres provides basic concurrency control but protects only the database's physical structure. It does not prevent multiple users from destroying the data entity structure. The data manager's session control provides the concurrency control necessary to protect the data entities' consistency. It accomplishes this by marking all needed entities at the beginning of a session. The data manager does not release the entities until the session is successfully terminated. Unlike large databases with many interactive users, the central System 690 database has a low volume of users at any one time and the likelihood of multiple users simultaneously trying to access the same data entity is very small.

Error Handling. Error handling was an important implementation issue with maintaining database consistency being the primary concern. Error handling by the data manager consists of three steps. The initial, and most important, is error detection. The next step is correcting any errors encountered. The final step is informing the user of the error and its cause.

Error detection is an extensive portion of the data manager. Each transaction request is validated for proper format and contents. If the transaction is an update, the standard file's description header is also validated for proper format, and, in the case of a file being checked-in,

a valid session identifier. If an error is detected during validation, the transaction is terminated. If no error is detected, the database updates begin.

Any errors encountered after request validation are caused by an invalid entity list in the standard data file or an Ingres error. An invalid entity list will be the most common error and is usually caused by the standard file being corrupted during a system-to-system file transfer. This type of error is detected during database update transactions. If the entity list contains all new entities, any entity which had been written to the database prior to the error is deleted. If the request is a session check-in, all database updates are reversed, leaving the database and session control files in the state prior to the attempted check-in. Ingres errors are a more serious problem because they indicate a possible system error. If an Ingres error is encountered, the data manager attempts to return the database to the state prior to the update request. This type of error may corrupt the database and must be handled carefully by the database administrator.

All errors are reported to the user. The data manager identifies the type of error encountered and the possible cause. This information is provided to a user in the .res file associated with each transaction.

The form of error handling used in the data manager is unforgiving. As soon as an error is detected, the database

is restored; if necessary, the error is reported; and the data manager then terminates. Database consistency was the driving concern and this consistency must be maintained, at the expense of extra effort by users and tool developers. This burden falls mostly on the tool developers in their work in generating a valid standard data file.

Test

The testing strategy used in developing the data manager occurred in four phases. These phases were unit testing, integration testing, validation testing, and system testing (13: 502). This strategy is shown in Figure 23.

Unit testing examined a module's interface, data structure integrity, boundary conditions, and error handling (13: 503-504). Each of these areas was tested using both test data and through normal use of the modules. Because of the extensive data passing between modules, the module's ability to maintain a structure's integrity was focused on. The module's error handling capability was also tested heavily because of the database integrity issue.

Integration testing was the next test phase. Integration testing focused on uncovering interface errors (13: 507). A bottom-up incremental integration test was used (13: 508). This method was selected because the data manager's low level modules contain the database update and retrieval routines. The successful implementation of the data manager depended on these modules working properly.

These lower modules were tested first to determine the feasibility of using a data definition table and generic database access routines.

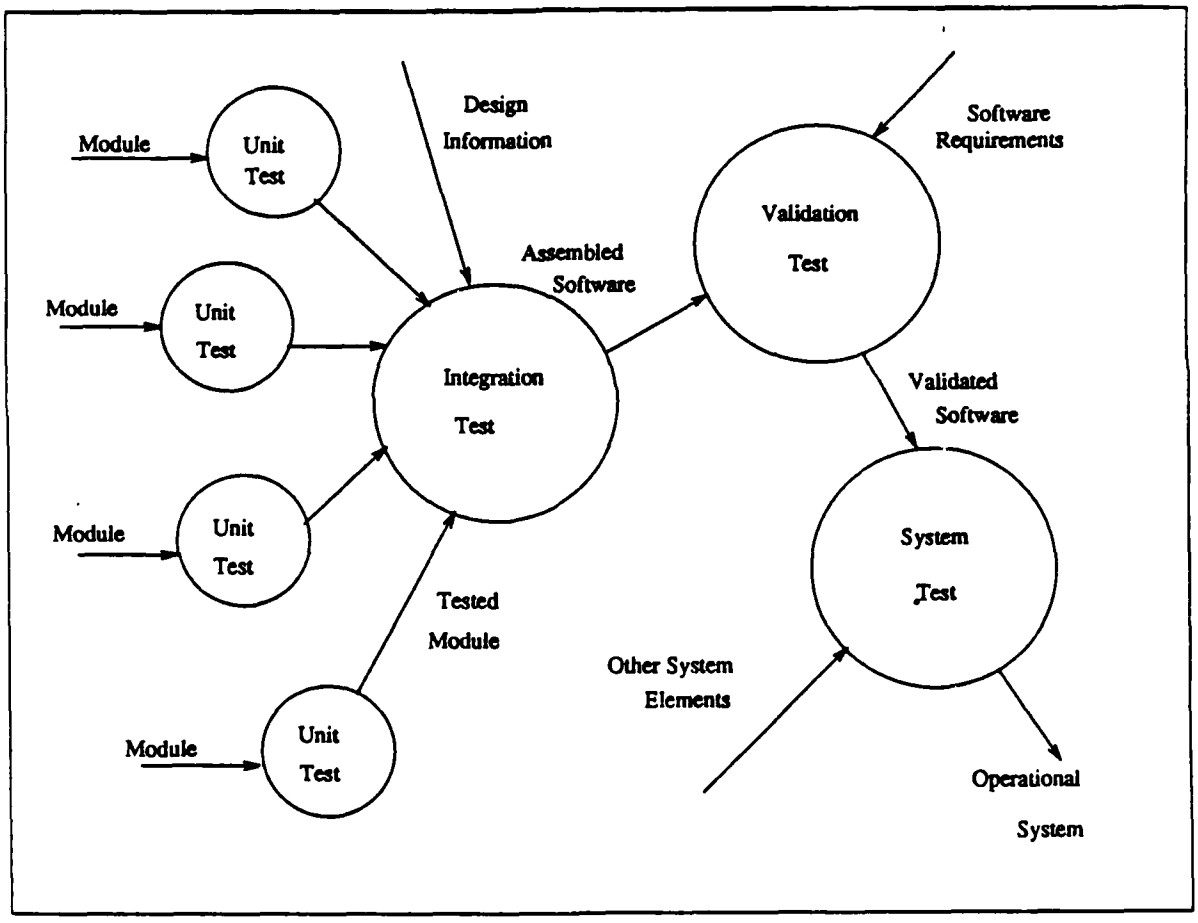


Figure 23. Software Testing Steps (13: 503)

Validation testing occurred next. This testing phase is concerned with the "does it work as expected" question (13: 514). The data manager validation test measured its ability to properly process data generated by the current SEL tools. The tools used in the test were the new SADT Editor (9) and the data dictionary editor. The SADT Editor files were successfully tested. The data dictionary file test required the development of a translator. The translator converts the standard data file for the design phase data entities to the data dictionary file format and vice-versa.

System testing is concerned with overall system issues, such as software and hardware compatibility, and usually involves different groups of individuals (13: 516). In this instance, the system issues were addressed in the validation tests.

Evaluation

The data manager was evaluated based on two criteria: how easy was it to use in integrating tools into System 690 and what were its performance characteristics. The integration evaluation was based on integrating a new tool into System 690 and integrating an existing tool. The performance evaluation was based on a series of tests measuring the time the data manager took to complete a specific transaction.

New Tool Integration. The integration of the standard data file, data manager, and a new tool (9) was deemed successful by the tool's developer. No significant errors were encountered in using the standard data file. The structure of the data element entries was easy to manipulate by the tool. Some of the entries within an element (ie. multi-line indicator) were not used but with each entry on a separate line, the unused entries could be easily skipped by the tool.

The overall effort of incorporating the standard data file was small. Most of the effort was needed in maintaining and generating the file description header information. The estimated effort to use the standard data file was 1% of the programming effort.

Existing Tool Integration. The existing tool chosen for integration with the data manager was the Data Dictionary (DD) editor. The DD editor provided an excellent opportunity to measure the integration effort because the DD editor's execution is directly related to its file format. For this reason, a translator was developed to convert the standard data file to a DD file and vice-versa. The DD editor also had to be modified to support using the standard data file.

The bulk of the work required to integrate the DD editor with the standard data file was in developing the translator. This translator was straightforward to design

and implement because the standard data file is already in the proper order so it is a one-to-one translation for each field.

The DD editor had to have a minor change made so that it could track the session identifier. This change was necessary to insure the session identifier remained with the associated file. The modification required the addition of only 10 lines of code and had no impact on the editor's performance.

The results of integrating the standard data file and the tools showed that the file lends itself to the System 690 tool structure and it supports both old and new tools. The ease of integration was a key requirement of the data manager and it was successfully met.

Performance. The standard data file was integrated with both new and old tools which was a basic requirement. Another requirement was for the data manager to provide an acceptable level of performance. This section evaluates the execution performance of the data manager.

To measure the data manager's performance a series of test jobs were run at various times of the day for a week on the two VAX 11/785 systems (ASC & SSC) available within AFIT. The job consisted of a multi-level retrieval, 1 level, and the immediate submittal of this session file for writing to the database. The resulting standard data file contained seven action entities and two object entities.

The jobs were run at two hour intervals from 0800 until 0200 to examine the data manager's performance under a wide variety of system loads.

Results. The performance parameters measured were the times required to perform the retrieval and the time required to perform the update. System load was based on the number of users on the system during data manager execution. A comparison was made between the number of users and active processes and they were found to be proportional.

The final results showed a surprising performance difference between the ASC and SSC computer systems. Although the two systems had approximately the same number of users, system configuration, and identical databases, the ASC executed the jobs, on average, twice as fast as the SSC. This difference was attributed to the SSC's job mix. The SSC is used in the AFIT Engineering School and processes more computational intensive jobs than the ASC. Because of the differences, the ASC results (Fig. 24) are presented.

The average time to perform the retrievals was 121 seconds with seven users on the system. The average time to perform the updates was 114 seconds. The SSC results, based on the best performance for each time period, were 280 seconds for the retrieval and 257 seconds to perform the update with an average of eight users on the system. The ASC times were improved to 114 seconds for the retrieval and

106 for the update when the worst performance time period was discarded. The time of day was 2400 when overnight processing was initiated. The worst performances observed were 200 seconds for the retrieval and 213 seconds for the update.

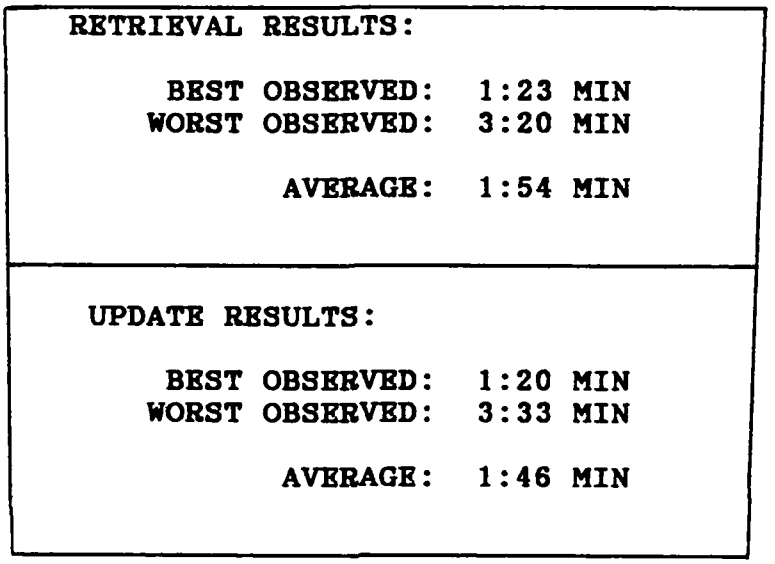


Figure 24. Data Manager ASC Performance Results

The retrieval consists of building the list of entities to be retrieved based on the parent and retrieving and writing the entities to the standard data file. These times were monitored internally by the data manager. The time required to build the entity list using the multi-level retrieval was important because this incurs much more overhead than a retrieval specifically listing the required entities. The average time to perform the multi-level list build was 40 seconds. Without this overhead the retrievals would finish in about 90 seconds, but the extra time is a

small penalty for the benefits derived. The user is assured of receiving all the existing entities and it saves him the trouble of typing in the entity list. A user could not manually build the list as fast as the data manager.

The update time will be proportional to the number of entities in the file. There is no way for the user to increase its performance, but less than two minutes to update the database is acceptable.

Summary

The data manager implementation was greatly affected by the AFIT computer environment which helped dictate the use of Unix, C, and Ingres. The capability for background processing in Unix was an important factor in developing the data manager interface. The interface was implemented to require minimal user interaction. The data manager implementation also placed a high degree of emphasis on error detection and recovery, with database consistency being the main concern.

The data manager was thoroughly tested beginning at the lowest levels and slowly integrated from bottom to top. The data manager was tested using test data and tool-generated data. The validation testing showed that the data manager was capable of processing tool requests and could successfully perform its expected functions.

The evaluation measured two key features of the data manager: the ease of integrating the standard data file and

tools and the performance of the data manager. The results of integrating the standard data file into both old and new tools showed the file to be easily integrated with minimal impact on the tool itself. The integration of the standard data file into existing tools will probably always require a translator be developed, but only a single translator is needed. Other tools do not have to build a translator to use another tool's data. This avoids the need for a "power set" (14) of interfaces.

The performance measurement showed an acceptable response time for performing updates and retrievals. The poor performance of the SSC was surprising, but it did prove the advantage of having a batch interface. If the data manager was implemented on a slow system, it could be executed in batch mode without the user having to wait 10 minutes for the transaction to finish.

VII. Conclusions and Recommendations

Conclusions

The purpose of this thesis was to implement a common database interface which integrates the separate tools in the AFIT Software Engineering Laboratory. This interface was required to not only integrate the existing tools but also support the addition of future tools to the SEL. The integrated tools would then be able to share a common database.

The method selected to implement the interface was the use of a standard data file and a data manager. The standard data file is used to transfer data between the tools and the data manager. The data manager performs all database updates and retrievals.

The standard data file has a standard structure that all the tools and the data manager can interpret. The flexibility of the standard data file is provided by being able to adjust the order and contents of the data elements in the file based on a tool's specific needs.

The data manager is the key component of the interface. Its key features are its ability to generically perform database updates and retrievals, provide session control, and support error recovery. The generic database access allows an existing tool's data file requirement to be easily incorporated with little or no programming changes. The

generic access also allows the addition of new tools into the SEL. The session control function performs basic data access control by allowing only an entity's owner to modify it. Session control also provides the librarian function of checking data entities in and out of the database and monitoring their status. The final function of the data manager was to provide error recovery. The data manager insures the logical structure of the data entities is maintained and that it can restore the database back to the state prior to the error.

The standard data file is an ASCII file which can used on all the current SEL workstations and those which are planned for later addition. The data manager is implemented in C and uses the Ingres DBMS. The data manager makes extensive use of Ingres to maintain the various tables it uses in performing its functions.

The implementation of the common database interface was evaluated to see if it met its integration and performance requirements. The standard data file was found to be easy to use by both old and new tools and well suited for its role in the interface. The data manager was evaluated to measure its execution performance. The evaluation found the data manager capable of being able to perform a typical database retrieval or update in under two minutes. This time is acceptable within the System 690 environment.

Overall, the implementation of the data manager and standard data file satisfied all the established requirements. A system which successfully integrates the existing System 690 tools was implemented and it was shown that this system supports the addition of new tools into System 690 and creates an integrated software engineering environment.

Recommendations

The implementation of the data manager presents the opportunity for several enhancements to the System 690 environment. Primarily these enhancements are modifications to the data manager or the development of database utilities. However, the data manager provides the means for new, more comprehensive and flexible tools to be added to System 690.

The data manager and standard data file need to be modified to allow multiple start/stop time entries. These multiple entries would support a more comprehensive tracking of tool usage patterns. Currently, only the last time a tool was used is passed to the data manager. This does not reflect true tool usage because the data is manipulated several times during a session. With only one entry, none but the last of these manipulations are retained in the database, providing a distorted view of tool usage.

In conjunction with the multiple start/stop time tracking capability, an enhanced performance measurement tool needs to be developed. The tool would monitor the

appropriate information (ie. user, tool id, transaction, results) to accurately measure tool and data manager usage and performance. As part of this tool, a history mechanism could be provided to monitor trends such as unbalanced tool usage and to identify and isolate the cause of consistent tool or data manager errors.

A standard data file print utility needs to be developed. The only means available to print data dictionary entries are to convert the standard data file into a data dictionary file and use an existing print program. The print utility needs to be able to print a standard data file for any phase and for multiple data entities in the file.

To support an integrated environment, a database consistency checking system needs to be developed. The data manager maintains consistency only at the entity level. A means to maintain design consistency within a phase and, more importantly, across phases is needed. This would prevent many of the design inconsistencies introduced during system development, especially in group projects.

To better support batch tool transaction requests, a file transfer system needs to be developed. A means to transfer files within System 690 without user interaction would provide the basis for a distributed design environment where a designer would only have to know how a tool works, not where or how its data files are used or stored.

Appendix A: Data Dictionary Database Relations and Data Dictionary Descriptions

The database relations for the data used in the data dictionary entries specified in the Software Development Documentation Guidelines and Standards (5) were developed by Thomas (16) and refined by Foley (4). This appendix provides the definitions of the data dictionary database relations and indicates the database relation and attribute for each data dictionary entry field.

KEY: a) The class of each relation is shown in parenthesis:
ie. activity (1) -- activity relation; Class 1

b) The type of each data dictionary field is provided in parenthesis:

(S) -- Single-line field
ie. NAME (S): name

(M) -- Multi-line field
ie. DESCRIPTION (M): desc line1
desc line2

(G) -- Group field
ie. ALIASES (G): aliasname
WHERE USED: where
COMMENT: comment

c) Each data dictionary field indicates the relation and attribute for the field and its key(s). The key is formed using the indicated attribute(s) and project.
ie. NUMBER: activity number aname

The relations and data dictionary entries are presented in the following order:

REQUIREMENTS PHASE:
- ACTIVITY -- Action Entity
- DATA ITEM -- Object Entity

DESIGN PHASE:
- PROCESS -- Action Entity
- PARAMETER -- Object Entity

IMPLEMENTATION PHASE:
- MODULE -- Action Entity
- VARIABLE -- Object Entity

ACTIVITY RELATIONS

```
activity (1)
     project      c12
     aname        c25
     number       c20
     status       c1

adesc (2)
     project      c12
     aname        c25
     line         i2
     description  c60

activityio (3)
     project      c12
     aname        c25
     diname       c25
     type         c11

aalias (4)
     project      c12
     aname        c25
     aliasname    c25
     comment      c60

ahierarchy (5)
     project      c12
     hianame      c25
     loaname      c25

areference (4)
     project      c12
     aname        c25
     reference    c60
     reftype      c25

ahistory (6)
     project      c12
     aname        c25
     version      c10
     date         c8
     author       c20
     comment      c60
```

ACTIVITY DATA DICTIONARY ENTRY

NAME (S): activity aname aname

TYPE: ACTIVITY

PROJECT (S): activity project aname

NUMBER (S): activity number aname

DESCRIPTION (M): adesc description aname line

INPUTS (M): activityio diname aname type (IN)

OUTPUTS (M): activityio diname aname type (OUT)

CONTROLS (M): activityio diname aname type (CON)

MECHANISMS (M): activityio diname aname type (MECH)

ALIASES (G): aalias aliasname aname

COMMENT: aalias comment aname

PARENT ACTIVITY (S): ahierarchy hianame loaname

REFERENCE (S): areference reference aname

REFERENCE TYPE: areference reftype aname

VERSION (S): ahistory version aname

VERSION CHANGES (S): ahistory comment aname

DATE (S): ahistory date aname

AUTHOR (S): ahistory author aname

DATA ITEM RELATIONS

```
dataitem (8)
     project      c12
     diname       c25
     datatype     c25
     low          c15
     hi           c15
     span         c60
     status       c1

didesc (2)
     project      c12
     diname       c25
     line         i2
     description  c60

divalueset (4)
     project      c12
     diname       c25
     value        c15

dihierarchy (5)
     project      c12
     hidiname     c25
     lodiname     c25

dialias (4)
     project      c12
     diname       c25
     aliasname    c25
     comment      c60
     whereused    c25

diref (4)
     project      c12
     diname       c25
     reference    c60
     reftype      c25

dihistory (6)
     project      c12
     diname       c25
     version      c10
     date         c8
     author       c20
     comment      c60
```

DATA ITEM DATA DICTIONARY ENTRY

NAME (S): dataitem diname diname

TYPE: DATA ELEMENT

PROJECT (S): dataitem project diname

DESCRIPTION (M): didesc description diname line

DATA TYPE (S): dataitem datatype diname

MIN VALUE (S): dataitem low diname

MAX VALUE (S): dataitem hi diname

RANGE (S): dataitem span diname

VALUES (M): divalueset value diname

PART OF (S): dihierarchy hidiname lodiname

COMPOSITION (M): dihierarchy lodiname hidiname

ALIASES (G): dialias aliasname diname

WHERE USED: dialias whereused diname

COMMENT: dialias comment diname

SOURCES (M): activityio aname diname type (OUT)

DESTINATIONS:

INPUT (M): activityio aname diname type (IN, MECH)

CONTROL (M): activityio aname diname type (CON)

REFERENCE (G): diref reference diname

REFERENCE TYPE: diref reftype diname

VERSION (S): dihistory version diname

VERSION CHANGES (S): dihistory comment diname

DATE (S): dihistory date diname

AUTHOR (S): dihistory author diname

PROCESS RELATIONS

process (1)

project	c12
prname	c25
number	c20
status	c1

prdesc (2)

project	c12
prname	c25
line	i2
description	c60

processio (3)

project	c12
prname	c25
paname	c25
direction	c4
type	c4

pralias (4)

project	c12
prname	c25
aliasname	c25
comment	c60

prcall (5)

project	c12
prcalling	c25
prcalled	c25

pralg (2)

project	c12
prname	c25
line	i2
algorithm	c60

prreference (4)

project	c12
prname	c25
reference	c60
reftype	c25

prhistory (6)

project	c12
prname	c25
version	c10
date	c8
author	c20
comment	c60

PROCESS DATA DICTIONARY ENTRY

NAME (S): process prname prname

PROJECT (S): process project prname

TYPE: PROCESS

NUMBER (S): process number prname

DESCRIPTION (M): prdesc description prname line

INPUT DATA (M): processio paname prname direction (IN) type (DATA)

INPUT FLAGS (M): processio paname prname direction (IN) type (FLAG)

OUTPUT DATA (M): processio paname prname direction (OUT) type (DATA)

OUTPUT FLAGS (M): processio paname prname direction (OUT) type (FLAG)

ALIAS (G): pralias aliasname prname

COMMENT: pralias comment prname

CALLING PROCESSES (M): prcall prcalling prcalled

PROCESSES CALLED (M): prcall prcalled prcalling

ALGORITHM (M): pralg algorithm prname line

REFERENCE (G): prreference reference prname

REFERENCE TYPE: prreference reftype prname

VERSION (S): prhistory version prname

VERSION CHANGES (S): prhistory comment prname

DATE (S): prhistory date prname

AUTHOR (S): prhistory author prname

PARAMETER RELATIONS

parameter (8)

project	c12
paname	c25
datatype	c25
low	c15
hi	c15
span	c60
status	c1

padesc (2)

project	c12
paname	c25
line	i2
description	c60

pavalueset (4)

project	c12
paname	c25
value	c15

pahierarchy (5)

project	c12
hipaname	c25
lopaname	c25

paalias (4)

project	c12
paname	c25
aliasname	c25
comment	c60
whereused	c25

paref (4)

project	c12
paname	c25
reference	c60
reftype	c25

pahistory (6)

project	c12
paname	c25
version	c10
date	c8
author	c20
comment	c60

papassed (4)

project	c12
paname	c25
prcalling	c25
prcalled	c25
direction	c4
iopaname	c25

PARAMETER DATA DICTIONARY ENTRY

NAME (S): parameter paname paname

PROJECT (S): parameter project paname

TYPE: PARAMETER

DESCRIPTION (M): padesc description paname line

DATA TYPE (S): parameter datatype paname

MIN VALUE (S): parameter low paname

MAX VALUE (S): parameter hi paname

RANGE OF VALUES (S): parameter span paname

VALUES (M): pavalueset value paname

PART OF (S): pahierarchy hipaname lopaname

COMPOSITION (M): pahierarchy lopaname hipaname

ALIAS (G): paalias aliasname paname

WHERE USED: paalias whereused paname

COMMENT: paalias comment paname

REFERENCE (G): paref reference paname

REFERENCE TYPE: paref reftype paname

VERSION (S): pahistory version paname

VERSION CHANGES (S): pahistory comment paname

DATE (S): pahistory date paname

AUTHOR (S): pahistory author paname

CALLING PROCESS (G): papassed prcalling paname

PROCESS CALLED: papassed prcalled paname

DIRECTION: papassed direction paname

I/O PARAM NAME: papassed iopaname paname

MODULE RELATIONS

```
module (1)
     project      c12
     modname      c25
     filename     c25
     number       c20
     status       c1

moddesc (2)
     project      c12
     modname      c25
     line         i2
     description  c60

modpass (3)
     project      c12
     modname      c25
     varname      c25
     type         c4

moduleio (3)
     project      c12
     modname      c25
     varname      c25
     direction    c8
     type         c12

modcall (5)
     project      c12
     modcalling   c25
     modcalled    c25

modalias (4)
     project      c12
     modname      c25
     aliasname    c25
     comment      c60

modreference (4)
     project      c12
     modname      c25
     reference    c60
     reftype      c25

modhistory (6)
     project      c12
     modname      c25
     version      c10
     date         c8
     author       c20
     comment      c60

modalg (2)
     project      c12
     modname      c25
     line         i2
     algorithm    c60
```

MODULE DATA DICTIONARY ENTRY

NAME (S): module modname modname

PROJECT (S): module project modname

TYPE: MODULE

NUMBER (S): module number modname

DESCRIPTION (M): moddesc description modname line

PASSED VARIABLES (M): modpass varname modname type (PASS)

RETURNS (M): modpass varname modname type (RET)

GLOBAL VAR USED (M): moduleio varname modname
direction (USED) type (GLOB)

GLOBAL VAR CHANGED (M): moduleio varname modname
direction (CHANGED) type (GLOB)

FILES READ (M): moduleio varname modname
direction (READ) type (FILE)

FILES WRITTEN (M): moduleio varname modname
direction (WRITTEN) type (FILE)

HARDWARE INPUT (M): moduleio varname modname
direction (IN) type (HARD)

HARDWARE OUTPUT (M): moduleio varname modname
direction (OUT) type (HARD)

CALLING MODULES (M): modcall modcalling modcalled

MODULES CALLED (M): modcall modcalled modcalling

ALIASES (G): modalias aliasname modname

COMMENT: modalias comment modname

REFERENCE (G): modreference reference modname

REFERENCE TYPE: modreference reftype modname

VERSION (S): modhistory version modname

DATE (S): modhistory date modname

AUTHOR (S): modhistory author modname

FILENAME (S): module filename modname

ALGORITHM (M): modalg algorithm modname line

VARIABLE RELATIONS

variable (8)

project	c12
varname	c25
datatype	c25
low	c15
hi	c15
span	c60
storetype	c12
status	c1

vardesc (2)

project	c12
varname	c25
line	i2
description	c60

varvalueset (4)

project	c12
varname	c25
value	c15

varhierarchy (5)

project	c12
hivarname	c25
lovarname	c25

varalias (4)

project	c12
varname	c25
aliasname	c25
comment	c60
whereused	c25

varpassed (3)

project	c12
varname	c25
modname	c25
direction	c4

varreference (4)

project	c12
varname	c25
reference	c60
reftype	c25

varhistory

project	c12
varname	c25
version	c10
author	c20
date	c8

VARIABLE DATA DICTIONARY ENTRY

NAME (S): variable varname varname

PROJECT (S): variable project varname

TYPE: VARIABLE

DESCRIPTION (M): vardesc description varname line

DATA TYPE (S): variable datatype varname

MIN VALUE (S): variable low varname

MAX VALUE (S): variable hi varname

RANGE OF VALUES (S): variable span varname

VALUES (S): varvalueset value varname

STORAGE TYPE (S): variable storetype varname

PART OF (S): varhierarchy hivarname lovarname

COMPOSITION (M): varhierarchy lovarname hivarname

ALIASES (G): varalias aliasname varname

WHERE USED: varalias whereused varname

COMMENT: varalias comment varname

PASSED FROM (M): varpassed modname varname direction (FROM)

PASSED TO (M): varpassed modname varname direction (TO)

REFERENCE (G): varreference reference varname

REFERENCE TYPE: varreference reftype varname

VERSION (S): varhistory version varname

DATE (S): varhistory date varname

AUTHOR (S): varhistory author varname

Appendix B: Standard Data File Format

Overview

The standard data file (SDF) is the interface between the tools and the data manager (DM). The data file consists of two parts, a file description header and data elements. The format and contents of these two components are examined. The overall file structure is also presented.

File Description Header

The file description header (Fig. 1) provides a full description of the file's data contents to both the tool and the data manager. The TOOL ID indicates the tool the data elements are formatted for and the PHASE and TYPE fields provide the type(s) of data elements contained in the file. This section examines the format of the file description header and establishes the acceptable field values. Figure 2 provides an example file header.

Field Values

The standard data file may be generated by either the data manager or a tool. The file's source can affect a field's contents. In those instances, the differences are pointed out.

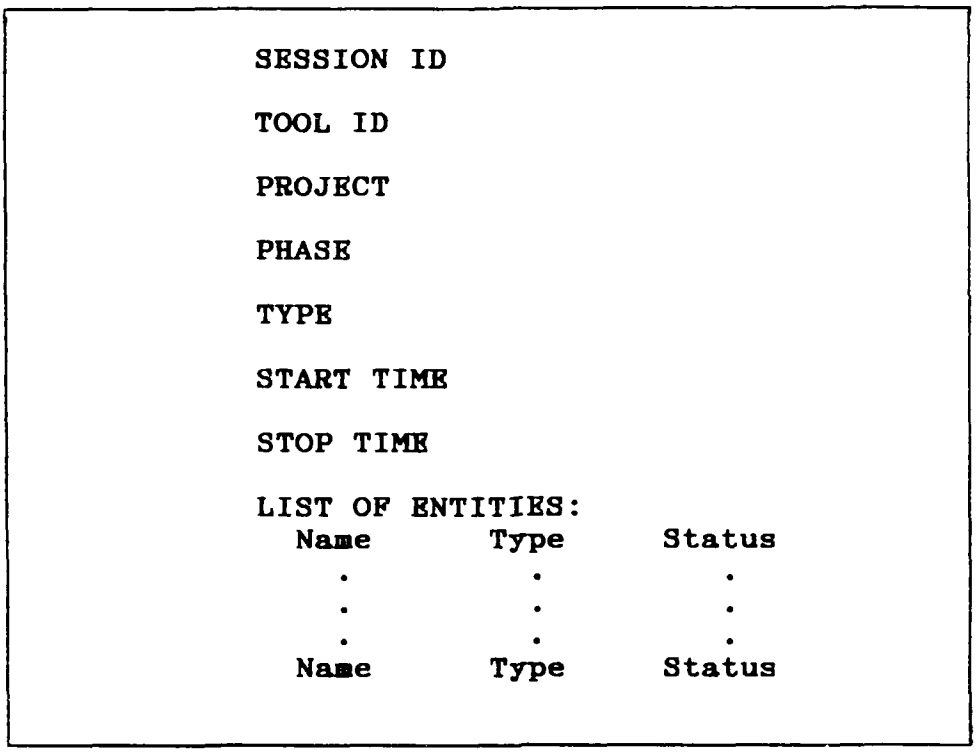


Figure 1. File Description Header Format

```
#@@BEGIN@@#
#@#HEADER BEGIN#@#
a08048710045
SADT
ECS SYSTEM
ACT
BOTH
000000
111111
Build Database              ACT       W
CRT Input                   OBJ       R
#@#HEADER END#@#
```

Figure 2. Example File Description Header

SESSION ID: The session identifier indicates the overall update status of the data contained in the file. There are three types of session id entries:

1) All data is in a retrieve status and may not be updated. (DM generated file)

ENTRY: "CONTAINS ONLY RETRIEVE ENTITIES"

2) All data entities are new and being submitted to the DM for the first time. (Tool generated file)

ENTRY: "SESSION CONTAINS ALL NEW RECS"

3) The data entities were retrieved from the database for update. (DM generated file)

ENTRY: "ammddyyhhmms" (session identifier)

a: Initial character must be "a"

The remaining fields are numeric. Any two position value (ie. mm, hh) which is less than 10 must use a 0 in the first position (ie. Jan->01).

mm: Month
dd: Day
yy: Year
hh: Hour (24 hour clock)
mm: Minutes
s: Seconds (10's position value)

TOOL ID: Contains the code identifying the tool. Used by the tool to insure the data is formatted for its use. Data manager uses the TOOL ID to help determine the format of the standard data file. Codes currently used:

1) SADT -- SADT Editor (Sun)

2) DD -- Data Dictionary Editor (Z-100)

PROJECT: Contains the project name of all the data entities in the file. May contain up to 12 characters.

PHASE: Identifies the phase of the data entities in the file. Codes currently used:

1) REQ -- Requirements Phase

2) DES -- Design Phase

3) CODE -- Code (Implementation) Phase

TYPE: Identifies the data type of the entities in the file. Codes used:

1) ACT -- Only ACTION entities are in the file.

2) OBJ -- Only OBJECT entities are in the file.

3) BOTH -- At least one ACTION entity and one OBJECT entity are contained in the file.

NOTE: These values may not be changed without extensive modification of the DM.

START/STOP TIME: Contains the start and stop times used in tracking a tool's usage. Entry format:

1) DM initialized value prior to tool usage:

START -> 000000
STOP -> 111111

2) Format of values provided by a tool after an update transactions:

ENTRY: "ddd mmm dd hh:mm:ss yyyy"

SAMPLE: "Wed Nov 18 12:12:03 1987"

ENTITY LIST ENTRY: Each entity entry consists of three fields. Each entry corresponds to an entity in the file. The order of the entities in the SDF MUST correspond to the entity list entry order. *NOTE: ACTION type entities MUST occur before object type entities when the file contains both types.

FORMAT:

ENTITY NAME:	Pos 0-24	(25 char)
BLANK:	Pos 25-29	(5 spaces)
ENTITY TYPE:	Pos 30-32	(3 char)
BLANK:	Pos 33-38	(6 spaces)
ENTITY STATUS:	Pos 39	(1 char)

ENTITY NAME: contains 1-25 characters

ENTITY TYPE: Codes used:

1) ACT -- ACTION entity

2) OBJ -- OBJECT entity

ENTITY STATUS: Indicates the update status of the entity. Codes used:

1) R -- Retrieve status, do NOT update. Can also be used by a tool to indicate an entity has not been changed during an update.

2) W -- Write entity to database. In new writes, W is the only acceptable status.

3) D -- Delete entity. May only be used in an update transaction where the entity has been checked-out. NOTE: A delete entity does NOT have a corresponding set of data elements in the file. Only entities in a W or R status have data elements in the file.

Data Elements

The data elements in the standard data file are grouped by data dictionary entry. The order of the elements are dictated by the Tool Data Definition Table order. This order must be maintained in files submitted by a tool to the DM for database update. The entity's name must be the first data element record in each entity. A second requirement is that every data element in a data dictionary entry be contained in the SDF, even if it has an empty ("blank") contents. The DM assumes a missing element was lost during file transfer and generates an error. The DM will also build the element with a blank contents field when the element's relation does not contain an entry.

Format

Figure 3 shows the format of a data element. Figure 4 provides an example data element.

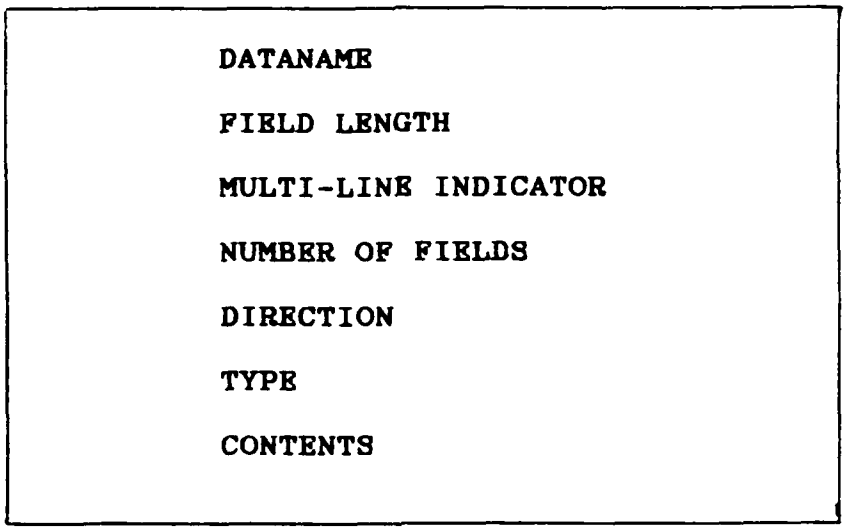


Figure 3. Data Element Record Format

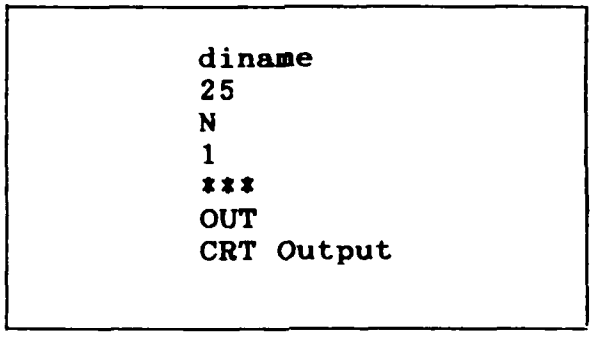


Figure 4. Sample Data Element Record

FIELD VALUES

DATANAME: Corresponds to the dataname maintained in the Tool Data Definition Table.

FIELD DESCRIPTION: Field length of the data element's contents field. Values: 1 - 60.

MULTI-LINE INDICATOR: Indicates whether the field consists of multiple lines (ie. description). Values Y or N.

NUMBER OF FIELDS: Indicates the number of fields associated with a group field.

```
ie. ALIAS:          (3)
       WHERE USED:  (2)
          COMMENT:  (1)
```

DIRECTION: Indicates the direction of the element. Used in instances where the same DATANAME applies to different data dictionary fields. Corresponds to the element's database direction value. Field contains "***" when it is unused.

```
EXAMPLE: INPUT FLAG
         OUTPUT DATA


DIRECTION: IN
           OUT
```

In the example above, both entries have the same DATANAME (paname). To differentiate between the two, DIRECTION, in conjunction with TYPE, is used to indicate which specific data dictionary field the element belongs.

TYPE: Indicates the data element's type. Corresponds to the element's database type value. May be used with the DIRECTION field or may be used by itself for elements which are dependent only on type, ie. INPUT, OUTPUT, CONTROL, MECHANISM in the acitivityio relation. Field contains "***" when unused.

CONTENTS: The database contents of DATANAME.

File Format

A complete example of the standard data file is shown in Figure 5. This example contains BOTH action and object entities. If the file contains only ACTION or OBJECT entities, the headers used for the other entity type are not included in the SDF.

```
#@@BEGIN@@#
#@#HEADER BEGIN#@#
a08048710045
SADT
ECS SYSTEM
ACT
BOTH
000000
111111
Build Database              ACT       W
CRT Input                   OBJ       R
#@#HEADER END#@#
###ACTION TYPE###
@##START##@
aname
N
1
***
***
Build Database
   o
   o  {Remaining data elements in Build Database}
   o
@##STOP##@
###ACTION END###
###OBJECT TYPE###
@##START##@
diname
N
***
***
CRT Input
   o
   o  {Remaining data elements in CRT Input}
   o
@##STOP##@
###OBJECT END###
#@@END@@#
```

Figure 5. Sample Data File Format

Appendix C: Data Manager Database Relations Definitions

Overview

The Data Manager (DM) uses various Ingres relations to control its data retrieval and update functions and to support session control. This appendix examines these relations and their formats. An overall example of how the relations are used is also provided.

Data Retrieval and Updates

The following relations are used to identify, retrieve, and update the data elements used in the standard data file (SDF).

Tool Description Table
Tool Data Definition Table
Entity Identification Table
Multi-level Transaction Table

Tool Description Table

The Tool Description table contains the name of the Tool Data Definition Table to use in generating and reading the SDF. The Tool Description Table contains a data definition table entry for each phase and type of data each tool uses. This section examines the Tool Description Table's Ingres relation (Fig. 1) and establishes its attribute formats.

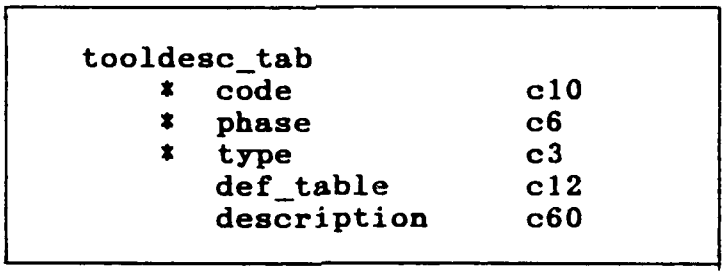


Figure 1. Tool Description Table Relation

Format

CODE: Tool Code. Code used to uniquely identify the tool, ie. DD.

PHASE: Phase of the data entity to be manipulated: REQ, DES, CODE

TYPE: Type of data to be manipulated: ACT or OBJ

DEF_TABLE: The relation name of the Tool Data Definition Table describing the data used by the tool in the indicated PHASE and TYPE.

DESCRIPTION: Means to provide additional information about the tool, ie. "SADT Editor - Sun Workstation REQ only".

Tool Data Definition Table

Contains the information necessary to retrieve data for a tool, generate the tool's SDF, read the tool's SDF, and perform the required updates. This section examines the table's Ingres relation (Fig. 2), specific requirements, and its attribute formats.

Requirements

1) RELATION NAME: Each phase and type of data entity used by a tool requires a separate data definition table to describe it.

Example: sadtdata -- SADT Data Item
sadtact -- SADT Activity

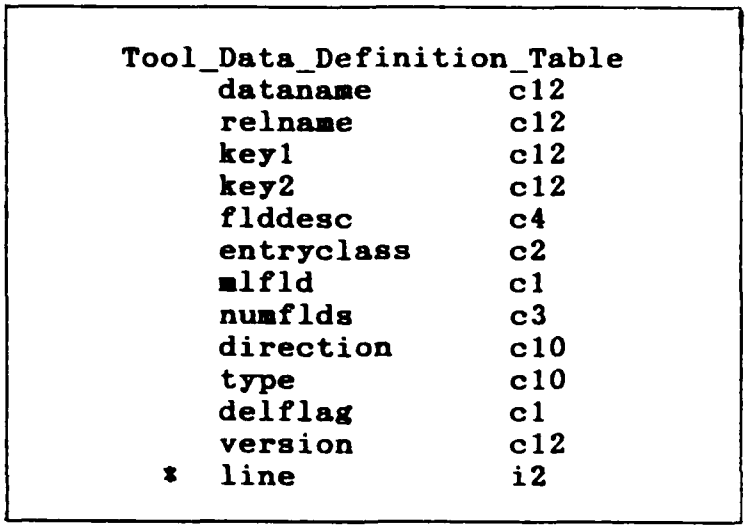


Figure 2. Tool Data Definition Relation

2) TABLE ENTRY ORDER: The table entry order is crucial. The table order dictates the order in which the DM writes data to the SDF and dictates the order the DM expects the data elements to be in when reading the SDF for updates.

The FIRST entry in the table must be the entity's attribute name. This is dictated by the DM which uses the field for validating updates and retrievals. The current first entries are the following:

REQ
ACT: activity aname
OBJ: dataitem diname

DES
ACT: process prname
OBJ: parameter paname

CODE
ACT: module modname
OBJ: variable varname

3) FIELD MARKERS: The table contains attributes (fields) that are not used by every element. Some of the attributes are very specific to support certain elements. These fields must contain "***" when they are unused because the DM checks for this value to determine if the attribute is required for a transaction.

Format

DATANAME: Data element name. The field has two purposes:

1) Attribute of the data element.
   ie. aliasname

2) Corresponds to dataname used in the SDF.

RELNAME: Relation name containing DATANAME.

KEY1: Key attribute name used to retrieve and update DATANAME. It is compared to the "entity name" being accessed.

```
ie.  range of r is relname
     retrieve (x = r.DATANAME)
     where (r.KEY1 = entity-name)
```

KEY2: Currently unused by any relation. Provided for future tools which require another key to access DATANAME. If unused, must contain "***".

FLDDESC: Field length of the Ingres DATANAME attribute. Also used as the FIELD LENGTH in the SDF.

ENTRYCLASS: Entry classification of DATANAME. The key field used by the DM to identify the retrieval and write routines to use with the element.

MLFLD: Multi-line field indicator. Contains either Y or N. Corresponds to MULTI-LINE INDICATOR in the SDF.

NUMFLDS: Number of fields in a single relation to be retrieved or written to the database.

```
ie. aliasname    3
    whereused    2
    comment      1
```

DM builds retrievals and updates using all three fields. It retrieves each table entry from n...1 for each database access. After retrieving the table entries, the DM then performs the retrieval or update.

DIRECTION: Direction used in certain relations (ie. processio - IN, OUT) to differentiate among the entries in the relation. For DATANAMEs which do not use DIRECTION the field contains "***".

NOTE: ALL relations which require a direction field MUST use the attribute name: direction.

TYPE: Type used in certain relations (ie. activityio - MECH, CON) to differentiate among the entries in the relation. For DATANAMEs which do not use TYPE the field contains "***".

NOTE: ALL relations which require a type field MUST use the attribute name: type.

DELFLAG: Delete Flag. Contains either Y or N. Used by the DM to control data deletions.

DELFLAG NOTE:

1) Certain data elements used in an entity do not get deleted when the entity is deleted because they are "owned" by another entity (ie. activityio elements used in a data item entry belong to an activity entity).

2) Only one element in a relation is marked for deletion. This deletes the entire tuple associated with the element.

ie.

aliasname	Y
whereused	N
comment	N

3) The DM performs two types of deletions:

a) Entity Delete -- Delete entire entity. Deletes all entries with DELFLAG = Y.

b) Update Delete -- Delete performed before writing an updated entity back to the database. For these deletes, DELFLAG = Y and VERSION = "***" are the only entries deleted. This prevents deleting version information associated with an entity.

VERSION: Version attribute name. Used with entries having an associated version (ie. ahistory).

LINE: Line number of the DATANAME entry in the table. The table is sorted by this field to help insure the data elements are retrieved in the proper order.

Entity Identification Table

The DM uses the Entity Identification table (Fig. 3) to determine the presence of a data entity in the database. Used during retrieval and update transaction verification.

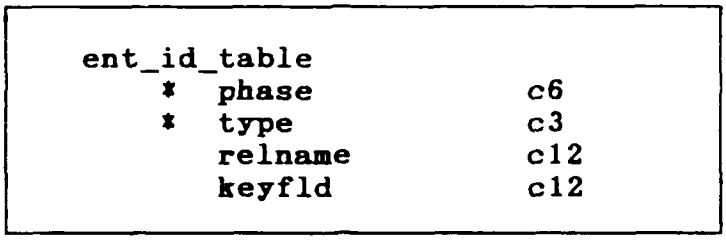


Figure 3. Entity Identification Table Relation

The following entries are currently used:

REQUIREMENTS PHASE

REQ	REQ
ACT	OBJ
activity	dataitem
aname	diname

DESIGN PHASE

DES	DES
ACT	OBJ
process	parameter
prname	paname

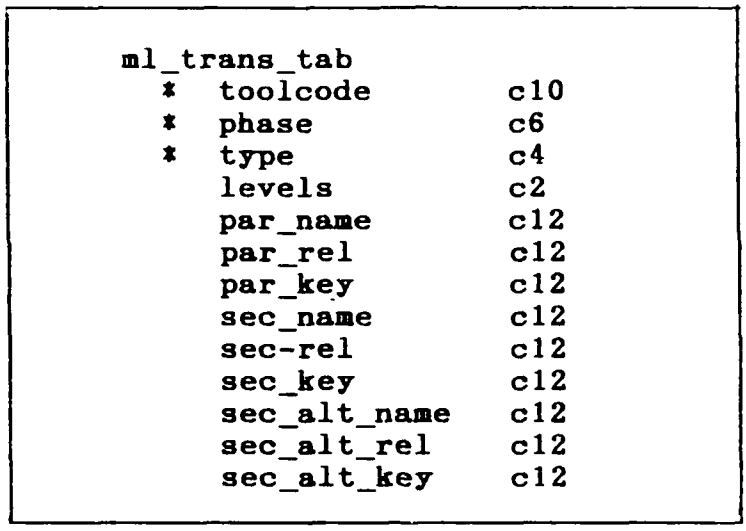
CODE PHASE

CODE	CODE
ACT	OBJ
module	variable
modname	varname

Multi-Level Transaction Table

The Multi-Level Transaction Table contains the information necessary to perform multi-level retrievals and

updates. This section examines the table's Ingres relation (Fig. 4) and attribute formats.



ml_trans_tab	
* toolcode	c10
* phase	c6
* type	c4
levels	c2
par_name	c12
par_rel	c12
par_key	c12
sec_name	c12
sec-rel	c12
sec_key	c12
sec_alt_name	c12
sec_alt_rel	c12
sec_alt_key	c12

Figure 4. Multi-Level Transaction Table Relation

Format

TOOLCODE: Tool code (ie. DD, SADT).

PHASE: Phase of data: REQ, DES, CODE

TYPE: Indicates the type of multi-level action to perform. The acceptable types are the following:

1) ACT -- Action entities only. Does not use the Secondary information.

2) OBJ -- Object entities only. Does not use Secondary information.

3) BOTH -- Uses both Parent and Secondary information.

NOTE: The values used in the Parent and Secondary entries may differ from the entries used to identify only the Action or Object entities.

LEVELS: The maximum number of levels of data a tool may retrieve. Values are 0-99. Level 0 identifies only the par_name and any sec_names and sec_alt_names associated with the parent. This is only for a BOTH request. An ACT or OBJ

Level 0 entry is the same as asking for only the single entity.

PAR_NAME: Parent Name. The field contains the entity name pointed to by the transaction file parent name.

PAR_REL: Parent Relation name.

PAR_KEY: Parent Key name. (Attribute name)

```
EXAMPLE:  par_name = loaname
          par_rel  = ahierarchy
          par_key  = hianame

          range of r is par_rel
          retrieve (r.par_name)
          where (r.par_key = trans_parent_name)
```

Secondary Entries

The secondary entries are used only in BOTH type transactions. These fields correspond to the data type opposite of the parent entity type.

SEC_NAME: Secondary Name. Identifies the entity name of an entity associated with one of the entities identified by PAR_NAME.

SEC_REL: Secondary Relation Name.

SEC_KEY: Secondary Key Name.

EXAMPLE: SADT Requirements Phase BOTH transaction

Parent	Secondary
name: loaname	name: diname
rel: ahierarchy	rel: activityio
key: hianame	key: aname

Procedures: Based on above example.

```
  Tool: SADT
 Phase: REQ
  Type: BOTH
Levels: 1
Parent: parent_name
```

1) Retrieve all activity names (loaname) pointed to by parent_name.

2) Repeat Step 1 until all requested levels have been retrieved or no new activities are identified. For these successive retrievals, use the activity names identified in the previous level for the key rather than the parent_name.

3) After all the activities are identified, the data items (diname) are retrieved. These entities are identified by using the identified activity names (aname = loaname) from Steps 1 and 2.

Secondary Alternate Entries

The alternate entries are provided to support the special cases where an entity may be identified in a relation separate from the secondary relation.

An example of this is shown below for the Design Phase.

EXAMPLE: Design Phase

Parent	Secondary	Alternate
name: prcalled	name: paname	name: iopaname
rel: prcall	rel: processio	rel: papassed
key: prcalling	key: prname	key: prcalling

Procedures: The procedures for alternate entities is the same as the above procedures but Step 3 is repeated using both the secondary and the alternate information.

Session Control

The following relations are used to support the DM session control functions.

Session Identification Table
Session Entity List
Entity Owner Table
Back-Up Directory Name

Session Identification Table

The Session Identification Table tracks each active session and maintains information describing the entities used in the session. This section examines the Session

Identification Table's Ingres relation (Fig. 5) and establishes the attribute formats.

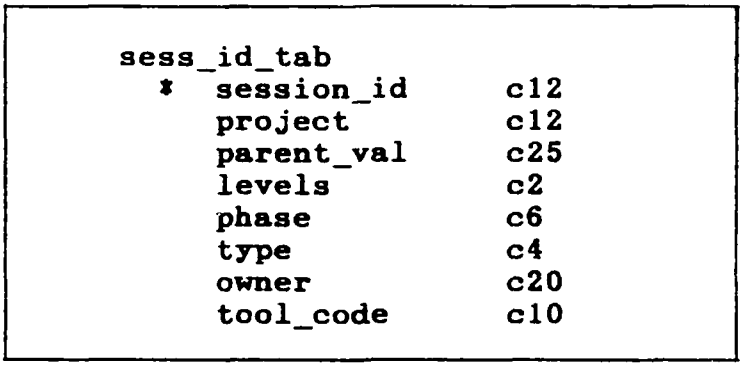


Figure 5. Session Identification Table Relation

Format

SESSION_ID: Session identifier in the SDF. DM uses this field to verify if a file submitted for update is an active session file.

PROJECT: Project name of the data contained in the SDF.

PARENT_VAL: Parent name used to retrieve the SDF entities. Blank if the entities were not identified using a multi-level retrieval.

LEVELS: Number of levels retrieved in a multi-level transaction. Blank if the SDF was not generated using a multi-level retrieval.

PHASE: Phase of the data in the SDF (REQ,DES,CODE).

TYPE: Type of the data in the SDF (ACT,OBJ,BOTH)

OWNER: Owner name provided in the transaction request.

TOOL_CODE: Tool code (ie. DD, SADT).

Session Entity List

The Session Entity List is a relation which monitors all checked-out data entities. This section examines the

Session Entity List's Ingres relation (Fig. 6) and establishes the attribute formats.

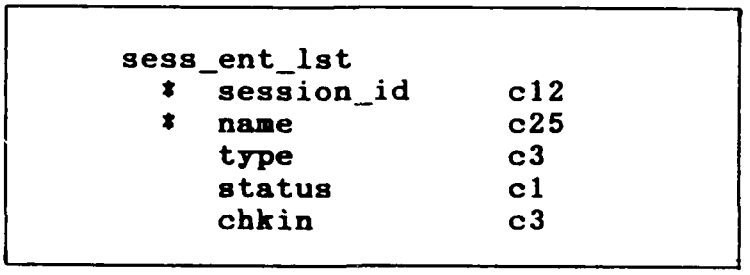


Figure 6. Session Entity List Relation

Format: The following entries are made for each checked-out entity. A single entity may have several entries in different sessions, which are identified by the session_id.

SESSION_ID: Session identifier used in the SDF and Session Identity Table.

NAME: Data entity name.

TYPE: Data type of the entity: ACT or OBJ.

STATUS: Identifies the update status of the data entity.

R -- Read only. No modifications allowed on entity.

W -- Write. Entity may be modified or deleted during the session.

CHKIN: Used by the DM when checking-in an updated session file. Values are "IN" or "".

Entity Owner Table

The Entity Owner Table is used by the DM to determine an entity's owner when being checked-out or deleted. This section examines the Entity Owner Table's Ingres relation (Fig. 7) and establishes the attribute formats.

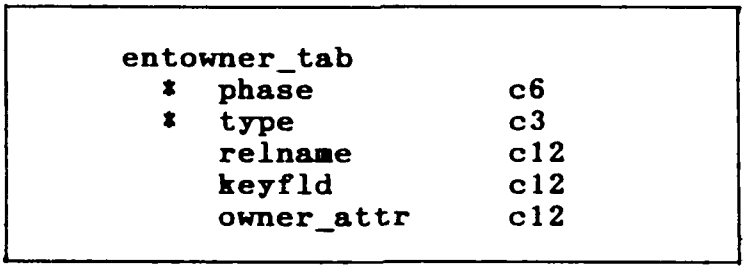


Figure 7. Entity Owner Table Relation

Format

PHASE: Phase of the data entity being requested for update. (REQ,DES,CODE)

TYPE: Type of the data entity being requested for update. (ACT, OBJ)

RELNAME: Relation name containing an entity's owner name. Currently, all entity owners are identified in the entity's History relation.

KEYFLD: Key attribute name in RELNAME which corresponds to the data entity name. Currently, Xname where X corresponds to the entity's type (ie. a - activity, pa - parameter).

OWNER_ATTR: Attribute name in RELNAME which contains the entity's owner name. The contents are compared against the transaction owner name. Only entities where these two values are equal may be modified. Currently, author is the attribute name which identifies an entity's owner.

Back-Up Directory Name

The Back-Up Directory Name relation (Fig. 8) consists of a single entry containing the name of the directory which contains all session back-up files created while using the database. These back-up files are used primarily for error recovery. The back-up file name is the same as the file's session identifier.

Figure 7. Entity Owner Table Relation

Format

DIR_NAME: Contains the FULL path name to the directory being used to store the back-up session files used in this database. This is a REQUIRED field. The DM will not run without a valid entry in this relation.

EXAMPLE: " /course/course/ee690/fa87/session_bkup.dir/"

NOTE: The blank before the first / must be included. No space can follow the last /.

Complete Sample Session

The following example shows the steps followed in a complete session.

Transaction Parameters:

Tool - SADT
Phase - REQ
Type - BOTH
Parent - Parent_Name
Levels - 1

Session Retrieval

1) Build list of activities and data items identified using the ml_trans_tab.

2) Use the ent_id_table to determine which of the identified entities exist in the database. Also identify the status of the existing entities.

3) Use entowner_tab to determine the owner of existing entities.

4) Add the valid entities, their status, and the associated session identifier to the sess_ent_lst.

5) Enter session in the sess_id_tab.

6) Use the tool data definition tables to retrieve the activities and data items which are written to the SDF.

7) Make back-up copy of the completed SDF using the directory contained in bkup_dirname.

Tool

1) Perform updates.

2) Submit modified file back to DM for database update.

Session Update

1) Verify the SDF session id is a currently active session using the sess_id_tab.

2) Identify new entities to be written to the database. Use ent_id_table to determine if the entity already exists. Error if entity name is already used.

3) Check for invalid update status using the sess_ent_lst (ie. a W status submitted for an entity checked-out in a R status). Error if invalid status detected.

4) Check all entities back in by setting sess_ent_lst.chkin = "IN".

5) Set all entity statuses to R which were not checked-in.

6) Perform updates using the appropriate tool data definition tables.

6a) Any errors encountered at this point require the use of the back-up session file to perform error recovery.

6b) Exit

7) Delete all sess_ent_lst entries identified by the SDF session id.

8) Delete sess_id_tab entry identified by the SDF session id.

9) Delete back-up session file.

Appendix D: User's Manual for the SEL Data Manager

Data Manager Overview

The Data Manager (DM) is a tool-database interface which permits any of the Software Engineering Laboratory (SEL) tools to use a central Ingres database. The basic requirements to use the DM are to have an account with Ingres access and access to the program dm. Your instructor will provide the appropriate login name and work directory necessary to use the DM.

The DM provides two basic functions: data retrieval and database update. These two functions consist of three steps: file transfer, DM transaction file generation, and DM execution. File transfer is necessary to move a tool generated data file to the DM system's working directory for updating the database and to move a DM generated tool file to the tool's system. Transaction file generation builds the instruction file used by the DM to perform its database transactions. The transaction file is built using an interactive menu. The remaining step is the execution of the DM using the generated transaction file.

The following sample scenario shows the steps which constitute a typical session.

> Sample Session: A user will execute dm to generate a transaction request to retrieve for update the needed data entity(s) from the database. On receipt of the request, the data manager will retrieve the data and provide this data back to the requestor in a session file using the provided session file name.
>
> The user will perform the necessary file transfer procedures to download the session file to the appropriate tool system. The user may manipulate this set of data entities by modifying or deleting the entities or by adding new entities to the file. When all changes have been made to the file, the session file must be transferred back to the DM system.

The user executes dm to generate a transaction request to update the database using the modified session file. On receipt of the request, the data manager will upload the data to the database. At the successful completion of the update, the session is terminated.

Each step in using the DM is discussed with sample sessions provided. Because each tool may reside on different systems, a separate attachment is provided for each tool describing the file transfer procedures to follow when using a particular tool. These attachments also include any special DM transaction requirements and limitations.

Please read the following instructions AND the appropriate tool attachment before using the DM.

File Transfer

The DM reads and generates tool files. For data retrievals, the DM generates a tool file which must be transferred to the tool system. You are responsible for performing these transfers.

Procedures: (Refer to the appropriate tool attachment)

RETRIEVALS: File transfer is the last step and occurs after the data manager finishes execution.

UPDATES: File transfer must be performed before executing the data manager.

Transaction Generation

The DM's execution is directed by a transaction file. This transaction file is generated using the interactive menu provided by dm.

Procedures:

1. For updates, transfer the tool file to the DM system.

2. Login to the DM system using the instructor provided login account and change directory to the proper working directory.

Example: (SSC system)

```
login: ee690
password:

SSC% cd fa87<CR>
SSC%
```

3. For updates, check that the file transferred in Step 1 is in the directory. (SSC% ls -l filename<CR> -> provides date the file was created to prevent using an old version.)

For retrievals, the DM creates a file to contain the retrieved data. If the file already exists, the new data overwrites the old contents. If the contents of any of the old files are needed, either provide the DM a different session file name for the new contents or rename the old file (SSC% mv oldfile newfile<CR>).

4. Execute the Data Manager:

```
SSC% dm<CR>
```

This generates the following menu:

```
******* BEGINNING OF DATA MANAGER MENU **********

                DATA MANAGER EXECUTION MENU

                1.  Build new transaction file for execution.
                2.  Use existing transaction file for execution.
                3.  Exit

                        ENTER CHOICE:

Enter transaction file name:

                        DATA MANAGER
                    TRANSACTION RECORD MENU


                        TOOL SELECTION
                    1.  Sun SADT Editor
                    2.  Data Dictionary Editor

                        ENTER CHOICE:[ ]

DATABASE NAME:[------------]

SESSION OWNER NAME:[--------------------]

                TRANSACTION INDICATOR SELECTION
                    1.  RETRIEVE DATA
                    2.  RETRIEVE DATA FOR UPDATE
                    3.  WRITE NEW DATA
                    4.  WRITE UPDATED DATA
                    5.  DELETE ENTITY
                    6.  ABORT SESSION
                    7.  EXIT TRANSACTION MENU

                        ENTER CHOICE:[ ]


SESSION IDENTIFICATION:[------------]


SESSION FILE NAME:[------------------------]

PROJECT:[------------]


                        TYPE SELECTION
                          1.  ACTIVITY
                          2.  OBJECT
                          3.  BOTH

                        ENTER CHOICE:[ ]
```

TRANSACTION ENTITY SELECTION
1. PARENT/LEVEL TRANSACTION
2. SPECIFIC ENTITIES

ENTER CHOICE:[ ]

PARENT NAME:[----------]

LEVELS:[--]

ENTITY NAME:[----------]

ENTITY TYPE
1. ACTION
2. OBJECT

ENTER CHOICE:[ ]

ADD ANOTHER ENTITY (Y or N):[ ]

TYPE OF DATA MANAGER EXECUTION

1. Background (Terminal remains available for other uses during DM execution)
2. Foreground (Terminal is used exclusively by DM and is unavailable during entire DM execution)
3. Exit

ENTER CHOICE:

********** END OF DATA MANAGER MENU **********

**NOTE: The menu presented above shows every field used for any type of transaction. Not every field is used for every transaction. Only the fields required for a specified TRANSACTION INDICATOR will be presented. You must provide a valid answer for any field presented for a particular transaction. The requirements and use of each field is presented below.

FIELD REQUIREMENTS: (NOTE: All fields are case sensitive)

TOOL SELECTION: Determines the format of the tool file the DM will read or generate.

DATABASE NAME: Database containing the tool data. Provided by course instructor.

SESSION OWNER NAME: Name used to determine user update and retrieval privileges, only the AUTHOR may modify an entity. Users should try to use the same name as the one used in the entity's AUTHOR field. (SUGGESTION: Maintain consistency of AUTHOR name of all entities used in a project by you and your team to permit easy update and retrieval.)

TRANSACTION INDICATOR SELECTION:

1. RETRIEVE DATA -- Retrieves data without checking for ownership. User is not permitted to modify any entities retrieved in this manner. Retrieved entities are stored in the session file specified by SESSION FILE NAME.

2. RETRIEVE DATA FOR UPDATE -- Retrieves data which may be updated. This transaction generates a session which tracks the entities which were checked out and their status. The status of each entity is maintained in the session file. The status is either Retrieve or Write. Only the entities in a Write status can be modified. The updated entities are resubmitted to the database using 4. WRITE UPDATED DATA. (**ATTENTION: Control files are generated for each session. No one can modify any entity you have checked-out in a Write status. If you do not want to submit the changes you made to the database, 6. ABORT SESSION may be used to release all your checked-out entities. To modify entities which belonged to an aborted session, the entities will need to be checked-out again.)

3. WRITE NEW DATA -- Writes all new entities to the database. This option is used when the tool file contains all new entities which are not currently in the database.

4. WRITE UPDATED DATA -- Used in conjunction with 2. RETRIEVE DATA FOR UPDATE. When all modifications, if any, have been made to the checked-out data, including addition of new entities, the session file is submitted to the DM for database updates.

5. DELETE ENTITY -- Used to delete entities which are no longer used or needed.

6. ABORT SESSION -- Used to release any entities which have been checked out. Requires the session identification of the checked-out data. The session id is shown in the .res file generated when using Background execution (option 1). For Foreground execution (option 2)

jobs, the session identification is the fourth line of the session file (SESSION FILE NAME with a .dbs extension).

7. EXIT TRANSACTION MENU -- Exit without processing the transaction.

SESSION IDENTIFICATION: Identification of the session to be aborted. Format: Field always starts with a small a. Example: SESSION IDENTIFICATION:[a10318712345] (*NOTE: The session identification is provided at session creation time. If this is unavailable, the session identification can be found on the fourth line of the session file.

SESSION FILE NAME: For retrievals, contains the name of the file to which you want the retrieved data to be written. For updates, contains the file name the DM expects to contain the entities necessary to perform the update transactions. Do NOT use the .dbs extension.

PROJECT: The project name of the entities. This is a very important field and MUST correspond to the project name of the entities to be retrieved or updated.

TYPE SELECTION:

1. ACTIVITY -- Indicates that only Activity entities are to be used. (ACTIVITY, PROCESS, MODULE)

2. OBJECT -- Indicates that only Object type entities are to be used. (DATA ITEM, PARAMETER, VARIABLE)

3. BOTH -- Indicates that both Activity and Object type entities are to be used.

TRANSACTION ENTITY SELECTION:

1. PARENT/LEVEL TRANSACTION -- Allows those tools which can process hierarchical entities a means to retrieve/delete the entities based on the parent name. The PARENT NAME and LEVELS are determined by the type of entities to be used and the tool. Refer to the specific tool attachment for additional information.

2. SPECIFIC ENTITIES -- Used to access the specific entities. This method is much faster than the PARENT/LEVEL method for accessing a limited number of entities.

ENTITY TYPE: Used when BOTH is chosen in the TYPE SELECTION. Allows the user to indicate the TYPE of each

entity. Answer the ADD ANOTHER ENTITY with a N when all the desired entities have been entered.

DATA MANAGER EXECUTION

The Data Manager provides two types of DM execution.

1. Background -- This type of execution does not use the terminal during processing and allows the user to either perform other transactions or logout. This is the recommended method for transactions containing 10 or more entities. It is also highly recommended for PARENT/LEVEL type transactions. The DM execution results are stored in trans_file_name.res. The .res file shows the entities which were successfully retrieved or written and contains any error messages which were generated during execution. This file contains the DM results only, do NOT confuse it with the SESSION FILE you provided. (HINT: The job number provided when the DM begins execution can be used to check the status of the execution.

Use the command: % ps alx jobnum<CR>)

2. Foreground -- This type of execution uses the terminal for displaying the DM results during execution. The terminal is unavailable for other use during the DM's execution. This method is useful when accessing a small number of entities, especially during retrievals because the user knows when the DM finishes. (HINT: If an error occurs and you need a hard copy of the error message, re-execute the DM using 1. Background mode.)

(**CAUTION: Because the database is being modified during any type of transaction, do NOT attempt to terminate the job. Improper termination could cause severe database inconsistencies.)

3. EXIT -- Do not execute the DM and return to the system (%).

SAMPLE SESSIONS

The following sample sessions show the menu options which must be completed for the various types of transactions available.

RETRIEVALS: (TRANSACTION INDICATOR 1 or 2)

```
% dm

                DATA MANAGER EXECUTION MENU

          1.  Build new transaction file for execution.
          2.  Use existing transaction file for execution.
          3.  Exit

                        ENTER CHOICE: 1

Enter transaction file name:  filename


                        DATA MANAGER
                  TRANSACTION RECORD MENU


                        TOOL SELECTION
                    1.  Sun SADT Editor
                    2.  Data Dictionary Editor

                        ENTER CHOICE:[1]

DATABASE NAME:[jtdb--------]

SESSION OWNER NAME:[Team 1A------------]

                TRANSACTION INDICATOR SELECTION
                    1.  RETRIEVE DATA
                    2.  RETRIEVE DATA FOR UPDATE
                    3.  WRITE NEW DATA
                    4.  WRITE UPDATE DATA
                    5.  DELETE ENTITY
                    6.  ABORT SESSION
                    7.  EXIT TRANSACTION MENU

                       ENTER CHOICE:[1]

SESSION FILE NAME:[level0----------------------]

PROJECT:[Homework 2--]

                        TYPE SELECTION
                          1.  ACTIVITY
                          2.  OBJECT
                          3.  BOTH

                       ENTER CHOICE:[3]
```

```
                TRANSACTION ENTITY SELECTION
                   1.  PARENT/LEVEL TRANSACTION
                   2.  SPECIFIC ENTITIES

                      ENTER CHOICE:[1]

PARENT:[Build Data Interface]

LEVELS:[1-]

SUCCESSFUL BUILD OF TRANSACTION FILE

                TYPE OF DATA MANAGER EXECUTION

   1.  Background (Terminal remains available for other uses
                   during DM execution)
   2.  Foreground (Terminal is used exclusively by DM and is
                   unavailable during entire DM execution)
   3.  Exit

                              ENTER CHOICE: 1

    Transaction results are in filename.res

    Batch Job is: [1] 19103

%
```

DATABASE WRITES: (TRANSACTION INDICATOR 3 or 4)

```
% dm

                DATA MANAGER EXECUTION MENU

          1.  Build new transaction file for execution.
          2.  Use existing transaction file for execution.
          3.  Exit

                        ENTER CHOICE: 1

Enter transaction file name:  filename


                        DATA MANAGER
                  TRANSACTION RECORD MENU


                        TOOL SELECTION
                    1.  Sun SADT Editor
                    2.  Data Dictionary Editor

                        ENTER CHOICE:[1]

DATABASE NAME:[jtdb--------]

SESSION OWNER NAME:[Team 1A------------]

              TRANSACTION INDICATOR SELECTION
                  1.  RETRIEVE DATA
                  2.  RETRIEVE DATA FOR UPDATE
                  3.  WRITE NEW DATA
                  4.  WRITE UPDATE DATA
                  5.  DELETE
                  6.  ABORT SESSION
                  7.  EXIT TRANSACTION MENU

                     ENTER CHOICE:[4]

SESSION FILE NAME:[level0----------------------]

PROJECT:[Homework 2--]

                      TYPE SELECTION
                        1.  ACTIVITY
                        2.  OBJECT
                        3.  BOTH

                     ENTER CHOICE:[3]

SUCCESSFUL BUILD OF TRANSACTION FILE
```

TYPE OF DATA MANAGER EXECUTION

1. Background (Terminal remains available for other uses during DM execution)
2. Foreground (Terminal is used exclusively by DM and is unavailable during entire DM execution)
3. Exit

ENTER CHOICE: 2

{** Results are printed to screen during execution **}

%

DELETIONS: (TRANSACTION INDICATOR 5)

```
% dm

                DATA MANAGER EXECUTION MENU

          1.  Build new transaction file for execution.
          2.  Use existing transaction file for execution.
          3.  Exit

                        ENTER CHOICE: 1

Enter transaction file name:  filename


                      DATA MANAGER
                TRANSACTION RECORD MENU


                      TOOL SELECTION
                  1.  Sun SADT Editor
                  2.  Data Dictionary Editor

                      ENTER CHOICE:[1]

DATABASE NAME:[jtdb--------]

SESSION OWNER NAME:[Team 1A------------]

              TRANSACTION INDICATOR SELECTION
                  1.  RETRIEVE DATA
                  2.  RETRIEVE DATA FOR UPDATE
                  3.  WRITE NEW DATA
                  4.  WRITE UPDATE DATA
                  5.  DELETE ENTITY
                  6.  ABORT SESSION
                  7.  EXIT TRANSACTION MENU

                     ENTER CHOICE:[5]

PROJECT:[Homework 2--]

                       TYPE SELECTION
                         1.  ACTIVITY
                         2.  OBJECT
                         3.  BOTH

                      ENTER CHOICE:[3]
```

```
                TRANSACTION ENTITY SELECTION
                   1.  PARENT/LEVEL TRANSACTION
                   2.  SPECIFIC ENTITIES

                      ENTER CHOICE:[2]

ENTITY NAME:[box 1----------------]

  ENTITY TYPE
  1.  ACTION
  2.  OBJECT

ENTER CHOICE:[1]

ADD ANOTHER ENTITY (Y or N):[Y]

ENTITY NAME:[data item 1-----------]

  ENTITY TYPE
  1.  ACTION
  2.  OBJECT

ENTER CHOICE:[2]

ADD ANOTHER ENTITY (Y or N):[N]


SUCCESSFUL BUILD OF TRANSACTION FILE

              TYPE OF DATA MANAGER EXECUTION

   1.  Background (Terminal remains available for other uses
                   during DM execution)
   2.  Foreground (Terminal is used exclusively by DM and is
                   unavailable during entire DM execution)
   3.  Exit

                          ENTER CHOICE: 2

   {**  Results are printed to screen during execution  **}

%
```

SESSION ABORT: (TRANSACTION INDICATOR 6)

```
% dm

                DATA MANAGER EXECUTION MENU

          1.  Build new transaction file for execution.
          2.  Use existing transaction file for execution.
          3.  Exit

                        ENTER CHOICE: 1

Enter transaction file name:  filename

                    DATA MANAGER
                TRANSACTION RECORD MENU

                    TOOL SELECTION
                1.  Sun SADT Editor
                2.  Data Dictionary Editor

                    ENTER CHOICE:[1]

DATABASE NAME:[jtdb--------]

SESSION OWNER NAME:[Team 1A------------]

            TRANSACTION INDICATOR SELECTION
                1.  RETRIEVE DATA
                2.  RETRIEVE DATA FOR UPDATE
                3.  WRITE NEW DATA
                4.  WRITE UPDATE DATA
                5.  DELETE ENTITY
                6.  ABORT SESSION
                7.  EXIT TRANSACTION MENU

                   ENTER CHOICE:[6]

SESSION IDENTIFICATION:[a10308709154]

SUCCESSFUL BUILD OF TRANSACTION FILE

            TYPE OF DATA MANAGER EXECUTION

   1.  Background (Terminal remains available for other uses
                   during DM execution)
   2.  Foreground (Terminal is used exclusively by DM and is
                   unavailable during entire DM execution)
   3.  Exit

                        ENTER CHOICE:
   {**  Results are printed to screen during execution  **}
```

This ends the sample sessions. Any additional questions should be directed to either the class DM manager or the instructor. The remainder of the User's Manual consists of the attachments for the separate SEL tools.

ATTACHMENT 1

User's Manual for the DATA MANAGER/SADT TOOL Interface

SADT Tool Overview

The SADT tool (SAtool) runs on the ZEUS Sun Workstation. The database this tool uses resides on the SSC. The following procedures are provided for this configuration. These instructions provide the basic procedures. For more specific information concerning Data Manager options, refer to the Data Manager User's Guide.

SADT Tool Operation

Tool Data File Generation:

The SAtool generates two types of files when building a SADT diagram. The files have a .dbs and a .gph extension. The .dbs file contains the diagram's entities in the DM file format. This is the file which is transferred to the SSC for database updates.

(**IMPORTANT: A .dbs file is created anytime you store the diagram using the SAVE FUNCTION. The file you submit to the DM for database update MUST be saved using the Save db option of the SAVE FUNCTION. This method performs consistency checks on the data and guarantees that the file is in the proper format for database transactions.)

SAMPLE DATABASE UPDATE PROCEDURES

STEP 1: Transfer the SAtool modified file to the Data Manager System. Assumes the two systems used are ZEUS and the SSC.

From ZEUS to SSC: (Method used in providing the DM data for Writing to the database.)

ZEUS% rcp sadtfile.dbs ssc:fa87/sadtfile.dbs

**NOTE: 1. sadtfile.dbs corresponds to the filename you used, it is NOT the mandatory filename.

2. Use the .dbs extension on the DM system

STEP 2: Login to the SSC using the account provided by the class instructor.

```
ZEUS% rlogin ssc

                    SSC Login Header Information
```

STEP 3: Change to the assigned working directory and check that the file was successfully transferred.

```
SSC% cd fa87
SSC% ls -l sadtfile.dbs
 -wrxr-xr-x  1 ee690    2199 Oct 30 09:34  sadtfile.dbs
SSC%
```

STEP 4: Begin Data Manager execution.

```
SSC% dm

              DATA MANAGER EXECUTION MENU

          1.  Build new transaction file for execution.
          2.  Use existing transaction file for execution.
          3.  Exit

                        ENTER CHOICE: 1

Enter transaction file name:  filename

                        DATA MANAGER
                   TRANSACTION RECORD MENU

                        TOOL SELECTION
                   1.   Sun SADT Editor
                   2.   Data Dictionary Editor

                        ENTER CHOICE:[1]

DATABASE NAME:[jtdb--------]

SESSION OWNER NAME:[Team 1A----------]  **Corresponds to
                                         author name


               TRANSACTION INDICATOR SELECTION
                    1.  RETRIEVE DATA
                    2.  RETRIEVE DATA FOR UPDATE
                    3.  WRITE NEW DATA
                    4.  WRITE UPDATE DATA
                    5.  DELETE ENTITY
                    6.  ABORT SESSION
                    7.  EXIT TRANSACTION MENU

                       ENTER CHOICE:[3]
```

```
SESSION FILE NAME:[sadtfile--------------------]
**Do NOT use .dbs extension, it is appended by the system

PROJECT:[Homework 2--]

                    TYPE SELECTION
                      1.  ACTIVITY
                      2.  OBJECT
                      3.  BOTH

                    ENTER CHOICE:[3]

SUCCESSFUL BUILD OF TRANSACTION FILE


              TYPE OF DATA MANAGER EXECUTION

   1.  Background (Terminal remains available for other uses
                   during DM execution)
   2.  Foreground (Terminal is used exclusively by DM and is
                   unavailable during entire DM execution)
   3.  Exit

                         ENTER CHOICE: 2

   {**  Results are printed to screen during execution  **}
   {**  Using foreground execution ties up the terminal **}
   {**  throughout execution.  You must wait until the  **}
   {**  DM finishes execution to do other processing or **}
   {**  to log out.                                     **}

              TRANSACTION COMPLETED
```

STEP 5: The Data Manager has finished. You may now continue working on the SSC or logout and return to ZEUS.

ADDITIONAL FILE INFORMATION: The following ls shows the two files used/created during an update transaction. The filename.ins was created during Transaction Record generation. It may be reused, if applicable, for option 2 of the DATA MANAGER EXECUTION MENU.

```
SSC% ls
       filename.ins            sadtfile.dbs
```

NOTE: Delete both these files when they are no longer needed. This will prevent excessive disk usage and prevent accidental usage of the files.

SAMPLE DATABASE RETRIEVAL PROCEDURES

The following session shows how to retrieve data from the database on the SSC and transfer these retrieved entities to ZEUS. [Assumes user is working on the ZEUS system. If starting from the SSC, follow instructions beginning at the first SSC prompt.]

STEP 1: Login to the SSC.

```
ZEUS% rlogin ssc

            SSC Login Header Information
```

STEP 2: Execute the Data Manager.

```
SSC% cd fa87
SSC% dm

 {**  DATA MANAGER EXECUTION MENU follows.  **}

  SESSION FILE NAME: sadtfile

  *(sadtfile can be any filename compatible with the
    SADT tool)

{**  The results of the retrieval are displayed  **}

SUCCESSFUL RETRIEVAL  <-- Indicates no errors occurred
```

STEP 3: Transfer the retrieved information to the SAtool system. *NOTE: The .dbs file extensions. This convention must be used.

```
SSC% rcp sadtfile.dbs zeus:fa87/sadtfile.dbs
```

STEP 4: Logout from SSC and return to the ZEUS system.

```
SSC%  logout

ZEUS%
```

The retrieved data has been transferred to the ZEUS system and may now be modified by the SAtool. Follow the Update Procedures presented above to transfer the data back to the SSC and update the database.

SPECIAL REQUIREMENTS

The SADT editor supports the use of hierarchical data. This permits a user to use the PARENT/LEVELS option in retrieving data. The SAtool is limited to only 1 level of data to be retrieved at one time. The options available are for a LEVEL of 0 or 1.

LEVEL 0: Retrieves only the activity entity associated with the PARENT and the data items used by this activity.

LEVEL 1: Retrieves the LEVEL 0 entities and the activity entities which are subordinate to the PARENT entity. All data items associated with any of the retrieved activity entities are also retrieved.

PARENT: The PARENT value corresponds the value used in the TITLE portion of the SADT diagram.

Appendix E: Tool Designer's Guide

Overview

The Data Manager (DM) supports the addition of a new tool to System 690 by adding the tool's data description information to the DM's control relations. The key relation(s) which must be built is the Tool Data Definition Table(s) which describes the data entity(s) the tool uses. This guide provides the procedures for defining a Tool Data Definition Table and describes the entry classes currently used by the DM. The procedures for defining the remaining control relations are also provided. A description of the Standard Data File's (SDF) use and the Transaction Request File's format are also provided.

NOTE: The Data Manager Database Relation Definitions, the Standard Data File Format, and the User's Manual for the SEL Data Manager should be available to provide additional relation and file information.

WARNING: ONLY the Database Administrator may execute the following commands because relation permissions are being set.

Tool Data Definition Table

The Tool Data Definition Table is the most important table used by the DM. It is also the most complex to create. This section examines how the table is created and provides a description of the entry classes currently recognized by the DM.

Table Creation

A tool requires that a Tool Data Definition Table be established for each data entity type it uses within a phase. Each of these tables must have a unique relation name. The format to use for the table's name is tool code and data type, ie. sadtdata, ddproc. This name may be no longer than 12 characters and must begin with a letter.

The following create command provides the Ingres instructions to create a new data definition table. The example is for the SADT Editor's data item entity.

```
create sadtdata (dataname   = c12,
                 relname    = c12,
                 key1       = c12,
                 key2       = c12,
                 flddesc    = c4,
                 entryclass = c2,
                 mlfld      = c1,
                 numflds    = c3,
                 direction  = c10,
                 type       = c10,
                 delflag    = c1,
                 version    = c12,
                 line       = i2)
\g
modify sadtdata to isam on line
\g
range of r is sadtdata
define permit retrieve on r to all
\g
```

Entry Class Definitions

The field formats and values for the Tool Data Definition Table are provided in the Data Manager Database Relation Definitions. The entry class determines the database access procedures to use in retrieving or updating a data element. All the data element's currently used in the six data dictionary entries can be described using only eight entry classes. New tools should be able to use the existing classes unless the tool requires the use of a relation(s) which is not currently defined in the database and shares no common characteristics with any of the current entry classes.

This section establishes the procedures to use in determining each field's value in relation to its use in the data element's entry class. The eight entry classes currently used by the DM are provided. The table entries used by the SADT Editor's activity (sadtact) and data item (sadtdata) are provided as examples. The Data Dictionary Editor's parameter (ddparam) table entries are also used.

CLASS 1: Describes an ACTION entity's identification relation, ie. activity, process, or module. For action entities, line 1 MUST be the entity's name field. The DM depends on this field being the first data element in the SDF for entity identification during update transactions.

```
append to sadtact (dataname   = "aname",
                   relname    = "dataitem",
                   key1       = "aname",
                   key2       = "\*\*\*",
                   flddesc    = "25",
                   entryclass = "1",
                   mlfld      = "N",
                   numflds    = "2",
                   direction  = "\*\*\*",
                   type       = "\*\*\*",
                   delflag    = "Y",
                   version    = "\*\*\*",
                   line       = 1)
\g
```

CLASS 2: Identifies data elements which are retrieved based on a line number, ie. description or algorithm. This class requires the relation's line attribute be named line.

```
append to sadtdata (dataname   = "description",
                    relname    = "didesc",
                    key1       = "diname",
                    key2       = "\*\*\*",
                    flddesc    = "60",
                    entryclass = "2",
                    mlfld      = "Y",
                    numflds    = "1",
                    direction  = "\*\*\*",
                    type       = "\*\*\*",
                    delflag    = "Y",
                    version    = "\*\*\*",
                    line       = 6)
\g
```

CLASS 3: Identifies data elements which use type, direction, or both of these attributes to identify the desired entries within a relation. The attribute names must be type and direction.

In the following example, please note:

1) Only the type field was used to identify the proper aname to access. The direction field contains "***" (the use of the \ is required by Ingres because * is a wild-card character) because it is not used to identify activityio elements. Other data dictionary entries (ie. varpassed) use only the direction field to differentiate between the relation's entries. Finally, entries such as

processio use both type and direction to identify the relation's entries. Always use the "***" to mark unused fields.

2) The delflag was set to "Y" for the following entry because it is the first of the activityio entities used by the DM. The three remaining activityio entries will use "N".

```
append to sadtact  (dataname   = "aname",
                    relname    = "activityio",
                    key1       = "diname",
                    key2       = "\*\*\*",
                    flddesc    = "25",
                    entryclass = "3",
                    mlfld      = "N",
                    numflds    = "1",
                    direction  = "\*\*\*",
                    type       = "IN",
                    delflag    = "Y",
                    version    = "\*\*\*",
                    line       = 3)
\g
```

CLASS 4: Identifies group fields, ie. alias, reference.

```
append to sadtdata (dataname   = "reference",
                    relname    = "diref",
                    key1       = "diname",
                    key2       = "\*\*\*",
                    flddesc    = "60",
                    entryclass = "4",
                    mlfld      = "N",
                    numflds    = "2",
                    direction  = "\*\*\*",
                    type       = "\*\*\*",
                    delflag    = "Y",
                    version    = "\*\*\*",
                    line       = 17)
\g
```

```
append to sadtdata (dataname   = "reftype",
                    relname    = "diref",
                    key1       = "diname",
                    key2       = "\*\*\*"
                    flddesc    = "25",
                    entryclass = "4",
                    mlfld      = "N",
                    numflds    = "1",
                    direction  = "\*\*\*",
                    type       = "\*\*\*",
                    delflag    = "N",
                    version    = "\*\*\*",
                    line       = 18)
\g
```

CLASS 5: Identifies hierarchical relations. These relation's delflag usage is critical to prevent deleting unowned relation entries. The higher order element (key1) has the delflag = "Y". The lower level entry is "owned" by a higher entity and does not necessarily belong to the entity being used. If both elements were deleted, the possibility exists to modify entities outside the current operation's domain.

```
append to sadtdata (dataname   = "hidiname",
                    relname    = "dihierarchy",
                    key1       = "lodiname",
                    key2       = "\*\*\*",
                    flddesc    = "25",
                    entryclass = "5",
                    mlfld      = "N",
                    numflds    = "1",
                    direction  = "\*\*\*",
                    type       = "\*\*\*",
                    delflag    = "N",
                    version    = "\*\*\*",
                    line       = 8)
\g
```

```
append to sadtdata (dataname   = "lodiname",
                    relname    = "dihierarchy",
                    key1       = "hidiname",
                    key2       = "\*\*\*",
                    flddesc    = "25",
                    entryclass = "5",
                    mlfld      = "N",
                    numflds    = "1",
                    direction  = "\*\*\*",
                    type       = "\*\*\*",
                    delflag    = "Y",
                    version    = "\*\*\*",
                    line       = 9)
\g
```

CLASS 6: Identifies the history relation. Unique aspect of these relations is their use of a version field. These are currently the only relations which have multiple versions. All versions are maintained in the database, but only the latest version is retrieved for tool use.

The DM recognizes relations which have multiple versions through the table's version entry. Only those version entries without "***" are recognized for update and retrieval purposes.

```
append to sadtdata (dataname   = "version",
                    relname    = "dihistory",
                    key1       = "diname",
                    key2       = "\*\*\*",
                    flddesc    = "10",
                    entryclass = "6",
                    mlfld      = "N",
                    numflds    = "4",
                    direction  = "\*\*\*",
                    type       = "\*\*\*",
                    delflag    = "Y",
                    version    = "version",
                    line       = 19)
\g
```

CLASS 7: Same data structure as CLASS 3 data BUT these entries are not updated during a session. These fields are included to provide additional data dictionary information. An example is the DESTINATION entries used in the Data Dictionary Data Item entry. The DM uses CLASS 3 retrieval procedures for generating the SDF entries, but uses CLASS 7 procedures when reading the updated SDF.

CLASS 8: Same function as CLASS 1 entries but the data elements making up this class occur throughout a dictionary entry. The entries are the OBJECT entities: data item, parameter, and variable. The definition table's first entry is still the identity relation's name field, but the remaining elements are used later in the SDF. (Currently used by the Data Dictionary Editor in the parameter entry.)

NUMFLD USE: In CLASS 8 relations, attributes do not occur contiguously in the data definition table. If only a single attribute is being accessed numfld = 1. If contiguous attributes are being accessed, use the numbering scheme n,n-1,...,1 for the numfld entries.

```
append to ddparam (dataname   = "paname",
                   relname    = "parameter",
                   key1       = "paname",
                   key2       = "\*\*\*",
                   flddesc    = "25",
                   entryclass = "8",
                   mlfld      = "N",
                   numflds    = "1",
                   direction  = "\*\*\*",
                   type       = "\*\*\*",
                   delflag    = "Y",
                   version    = "\*\*\*",
                   line       = 1)
\g

{Description is the second entry in the table.}

append to ddparam (dataname   = "datatype",
                   relname    = "parameter",
                   key1       = "paname",
                   key2       = "\*\*\*",
                   flddesc    = "25",
                   entryclass = "8",
                   mlfld      = "N",
                   numflds    = "4",
                   direction  = "\*\*\*",
                   type       = "\*\*\*",
                   delflag    = "N",
                   version    = "\*\*\*",
                   line       = 3)
\g
```

FUTURE CLASSES: New DM entry classes will be for those elements which are used in a manner different from those identified or whose relations use a different access method. The best guide for adding a new entry class will be the existing classes. The Class 8 entries will probably provide

the best starting point for identifying the needed table field entries. Warning: Be very careful in setting the delflag values. Improper setting of the flag can cause inconsistencies both in the entity and the entire project.

To add a new class, the DM will have to be modified. The DM is structured so only the class_X_retrieval and class_X_write modules will have to be added. This is a programming effort which requires knowledge of both C and EQUEL. As above, the code used for the other classes provides an excellent guide for developing the new code.

Other Control Relations

The remaining control relations support the DM either in performing database accesses or providing session control. This section provides the Ingres commands to create the relations and a sample relation entry.

Tool Description Table

The Tool Description Table identifies the Tool Data Definition Table that a tool uses in writing or reading the tool's SDF. Note: There may be similarities between the data definition table requirements for different tools using the same data dictionary data and the same table could satisfy both tools' needs. This is highly discouraged because a change in one tool's data needs could adversely impact the other tools using the same table.

Create: Create the tool description table. The only field which does not require a value to be assigned is the description field. This field is provided to allow the database administrator to further identify a description table and its user.

```
create tooldesc_tab (code        = c10,
                     phase       = c6,
                     type        = c3,
                     def_table   = c12,
                     description = c60)
\g
modify tooldesc_tab to isam on code,
                               phase,
                               type
\g
range of r is tooldesc_tab
define permit retrieve on r to all
\g
```

Sample Entries: The following samples show how the data definition tables containing the two data types used by the SADT Editor are added to the tool description table.

```
append to tooldesc_tab (code        = "SADT",
                        phase       = "REQ",
                        type        = "ACT",
                        def_table   = "sadtact",
                        description = "Johnson's SADT tool")
\g
append to tooldesc_tab (code        = "SADT",
                        phase       = "REQ",
                        type        = "OBJ",
                        def_table   = "sadtdata",
                        description = "Johnson's SADT tool")
\g
```

Session Identification Table

The Session Identification Table contains the information the DM used to create the SDF containing the indicated session. The table tracks all active sessions. The table's contents are manipulated only by the DM.

Create: Create the session identification table.

```
create sess_id_tab (session_id = c12,
                    project    = c12,
                    parent_val = c25,
                    levels     = c2,
                    phase      = c6,
                    type       = c4,
                    owner      = c20,
                    tool_code  = c10)
\g
modify sess_id_tab to isam on session_id
\g
range of r is sess_id_tab
define permit all on r to all
\g
```

Entity Identification Table

The Entity Identification Table identifies the relation name and key fields necessary to check for an entity's existence and write status. Every data dictionary entry type has a corresponding entry in the identification table. The sample shows entries for the requirements phase. Similar entries are also required for the design and implementation phases.

CREATE: Create the entity identification table.

```
create ent_id_table (phase   = c6,
                     type    = c3,
                     relname = c12,
                     keyfld  = c12)
\g
modify ent_id_table to isam on phase,
                               type
\g
range of r is ent_id_table
define permit retrieve on r to all
\g
```

SAMPLE ENTRIES:

```
append to ent_id_table (phase   = "REQ",
                        type    = "ACT",
                        relname = "activity",
                        keyfld  = "aname")
\g
append to ent_id_table (phase   = "REQ",
                        type    = "OBJ",
                        relname = "dataitem",
                        keyfld  = "diname")
\g
```

Entity Owner Table

The Entity Owner Table identifies the relation name and key fields necessary to identify an entity's owner. Like the Entity Identification Table, each data dictionary entry type has a corresponding table entry.

CREATE:

```
create entowner_tab (phase      = c6,
                     type       = c3,
                     relname    = c12,
                     keyfld     = c12,
                     owner_attr = c12)
\g
modify entowner_tab to isam on phase,
                               type
\g
range of r is entowner_tab
define permit retrieve on r to all
\g
```

SAMPLE ENTRIES:

```
append to entowner_tab (phase      = "REQ",
                        type       = "ACT",
                        relname    = "ahistory",
                        keyfld     = "aname",
                        owner_attr = "author")
\g
append to entowner_tab (phase      = "REQ",
                        type       = "OBJ",
                        relname    = "dihistory",
                        keyfld     = "diname",
                        owner_attr = "author")
\g
```

Back-Up Directory Name

The Back-Up Directory Name relation contains a single entry naming the directory name to be used by all users of the database for storing their back-up session files. These files are created by the DM during Update Retrieval transactions and used by the data manager during Update Write error recovery.

*NOTE: The table contains a single entry which establishes the full path name (from the root) for storing back-up session files. This entry has a very specific format which must be followed. The format of the entry is:

Format: " /dir/.../.../dir/bkup.dir/"

Example: " /course/course/ee690/fa87/"

Notice the first position before the "/" is a space. This is required by the DM. If the "/" is accidentally used in the first position, the DM will encounter errors in trying to create and delete the back-up session files. Also the string ends with a "/" without a trailing space. This format must be met exactly or none of the backup routines will work, preventing error recovery during session update transactions.

CREATE:

```
create bkup_dirname (dir_name = c100)
\g
range of r is bkup_dirname
define permit retrieve on r to all
\g
```

SAMPLE ENTRY:

```
append to bkup_dirname
      (dir_name = " /course/course/ee690/fa87/bkup.dir/")
\g
```

Multi-Level Transaction Table

The Multi-Level Transaction Table contains the relations and keys necessary to perform hierarchical database retrievals and deletes based on a parent name and the indicated number of levels. This table contains the entries for all tools using the database. The entries are tool, phase, and type dependent. Each tool which supports multi-level transactions has an entry(s). The entries identify whether to retrieve only ACT or OBJ entities or BOTH entity types. Notice that if a tool can manipulate both a single entity type and both entity types, the relations used for the parent identification may differ.

CREATE:

```
create ml_trans_tab (toolname     = c10,
                     phase        = c6,
                     type         = c4,
                     levels       = c2,
                     par_name     = c12,
                     par_rel      = c12,
                     par_key      = c12,
                     sec_name     = c12,
                     sec_rel      = c12,
                     sec_key      = c12,
                     sec_alt_name = c12,
                     sec_alt_rel  = c12,
                     sec_alt_key  = c12)

\g
modify ml_trans_tab to isam on toolname,
                               phase
\g
range of r is ml_trans_tab
define permit retrieve on r to all
\g
```

SAMPLE ENTRIES:

```
append to ml_trans_tab (toolname      = "DD",
                        phase         = "DES",
                        type          = "BOTH",
                        levels        = "1",
                        par_name      = "prcall",
                        par_rel       = "prcalling",
                        par_key       = "prcalled",
                        sec_name      = "processio",
                        sec_rel       = "prname",
                        sec_key       = "paname",
                        sec_alt_name  = "papassed",
                        sec_alt_rel   = "prcalling",
                        sec_alt_key   = "iopaname")
\g

append to ml_trans_tab (toolname      = "DD",
                        phase         = "DES",
                        type          = "OBJ",
                        levels        = "1",
                        par_name      = "pahierarchy",
                        par_rel       = "hipaname",
                        par_key       = "lopaname",
                        sec_name      = "\*\*\*",
                        sec_rel       = "\*\*\*",
                        sec_key       = "\*\*\*",
                        sec_alt_name  = "\*\*\*",
                        sec_alt_rel   = "\*\*\*",
                        sec_alt_key   = "\*\*\*")
\g
```

Standard Data File

The SDF requirements are identified in the Standard Data File Format. The key aspects of the SDF are:

1) The entities must be ordered in the file according to the data definition table order. The tool may not alter this order.

2) The entity order is action entities then object entities.

3) Entities in a D (delete) status do not have a corresponding entry in the data elements.

4) The first DATANAME in any entity must correspond to that entity's identifying attribute, ie. aname, paname.

5) Every DATANAME in the tool's data definition table must be included in the SDF. The DM expects these elements to be present and generates an error if any are missing. The DM does not write the "blank" contents to the database. On retrievals, the DM will automatically generate a "blank" contents field for the DATANAME.

Transaction Request File Format

The Transaction Request File (TRF) format is provided for tool designer's whose tool can generate a batch transaction. This discussion shows the field formats but does not discuss their role. For this information, refer to the User's Manual for the SEL Data Manager.

FORMAT

The following format shows the order of entries in a transaction request file. The contents of these fields are transaction dependent. Transactions fall into four general categories: retrievals, writes, deletes, and session aborts. Sample entries for these four transaction categories are also presented.

FILENAME FORMAT REQUIREMENTS: Two types of files are used in interacting with the DM. The files are the transaction request file and the SDF. The naming convention used is the following:

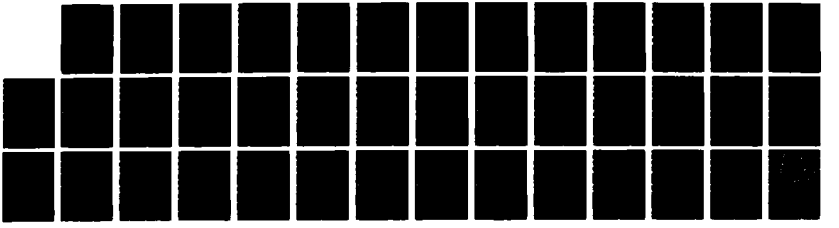
SDF -- filename.dbs (Must have .dbs extension)

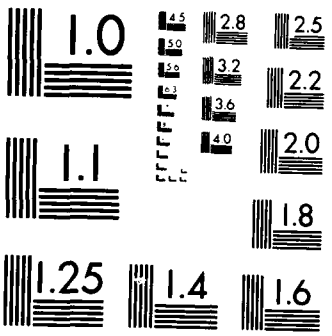
TRF -- filename.ins (Must have .ins extension)

GENERAL FORMAT: A transaction file contains the entries shown below, but as stated, not all the fields are used for every transaction. For the unused fields, the field contains "***". The use of this filler value can be seen in the transaction samples.

AD-A189 628 COMMON DATABASE INTERFACE FOR HETEROGENEOUS SOFTWARE ENGINEERING TOOLS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 3/3

UNCLASSIFIED T D CONNALLY DEC 87 AFIT/GCS/ENG/87D-8 F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

GENERAL FORMAT SAMPLE:

```
@@@BEGIN@@@
TOOL IDENTIFICATION
DATABASE NAME
OWNER NAME
PHASE
TRANSACTION INDICATOR
SESSION IDENTIFIER
SESSION FILE NAME
PROJECT NAME
ENTITY TYPE
PARENT NAME
LEVELS
LIST OF ENTITIES:
Name        Type
 o            o
 o            o
@@@END@@@
```

RETRIEVAL FORMAT: The retrieval transactions are of two types: retrieve only (R) and retrieve for update (UR). Two examples are provided, the first uses a multi-level retrieval and the second identifies the specific entities to be retrieved.

RETRIEVAL SAMPLE 1: Multi-Level Retrieval

```
@@@BEGIN@@@
SADT
sadtdb
author_name
REQ
UR
***
session_file_name.dbs
project_nm
BOTH
parent_name_for_ret
1
@@@END@@@
```

RETRIEVAL SAMPLE 2: Specific Entities

```
@@@BEGIN@@@
SADT
sadtdb
author_name
REQ
R
***
session_file_name.dbs
project_nm
BOTH
***
0
ACT_Entity_Name                    ACT
OBJ_Entity_Name                    OBJ
@@@END@@@
```

WRITE FORMAT: The write transactions are of two types: new writes (W) and write with update (UW). Only one example is provided because the formats of the two transactions are the same except for the transaction indicator code.

WRITE SAMPLE: Write with Update

```
@@@BEGIN@@@
SADT
sadtdb
author_name
REQ
UW
***
session_file_name.dbs
project_nm
BOTH
***
0
@@@END@@@
```

DELETE FORMAT: A delete transaction (D) allows the tool to identify the entity(s) to be deleted without having to build a SDF. The delete transaction supports deleting either explicitly named entities or via a multi-level transaction.

DELETE SAMPLE:

```
@@@BEGIN@@@
DD
dddb
author_name
DES
D
***
***
project_nm
OBJ
***
0
Entity_to_be_deleted          OBJ
@@@END@@@
```

ABORT FORMAT: An abort transaction (A) allows the tool to abort a session without checking the SDF back-in. This is provided in case the SDF is lost or corrupted. The session identification code is required for this transaction. It is available in the results (TRFname.res) file which is generated during batch transactions.

ABORT SAMPLE:

```
@@@BEGIN@@@
DD
dddb
author_name
DES
A
a08048712325
***
***
***
***
0
@@@END@@@
```

Appendix F: System Configuration Guide

This guide provides the configuration of the Data Manager (DM), Interactive Transaction Menu, and Data Dictionary-Data Manager File Translator programs.

Data Manager

The Data Manager code modules use the following include files:

```
 define.h -- constants definitions
datadef.h -- global data structures
ingdata.h -- global Ingres data structures
```

The DM consists of the following modules:

```
build_parent_list.o          class_7_write.o
class_1_ret.o                class_8_ret.o
class_1_write.o              class_8_write.o
class_2_ret.o                dm_main.o
class_2_write.o              error_routines.o
class_3_ret.o                proglib.o
class_3_write.o              retrieve_driver_mod.o
class_4_ret.o                rmain.o
class_4_write.o              session_ret_hdr_mod.o
class_5_ret.o                session_wrt_hdr_mod.o
class_5_write.o              trans_build_edit.o
class_6_ret.o                wmain.o
class_6_write.o              write_driver_mod.o
```

The DM code can be broken down according to the following functions:

Main

dm_main.o

Transaction Handling

trans_build_edit.o build_parent_list.o

Retrievals

```
rmain.o                          session_ret_hdr_mod.o
retrieve_driver_mod.o            class_1_ret.o
class_2_ret.o                    class_3_ret.o
class_4_ret.o                    class_5_ret.o
class_6_ret.o                    class_8_ret.o
```

Updates

```
wmain.o                          session_wrt_hdr_mod.o
 write_driver_mod.o               class_1_write.o
class_2_write.o                  class_3_write.o
class_4_write.o                  class_5_write.o
class_6_write.o                  class_7_write.o
class_8_write.o
```

Error Recovery

```
error_routines.o
```

Library Routines

```
proglib.o
```

Data Manager Generation

The DM is generated using the Unix make command. The DM_Makefile contains the make instructions. The script make.dm is used. The make.dm script and a sample DM_Makefile entry are presented. The complete DM_Makefile is provided in Attachment 1.

make.dm

```
echo 'make -f DM_Makefile'
make -f DM_Makefile
```

Sample DM Makefile Entry

```
dm_main.o : dm_main.body define.h datadef.h ingdata.h
     - rm dm_main.q
     - rm dm_main.c
     cat ingdata.h dm_main.body >> dm_main.q
     equel dm_main.q
     cc -c dm_main.c
```

Interactive Transaction Menu

The transaction menu program provides the interactive menu which generates the transaction request file.

Source: trans_menu.body

Compile and Link Script: make.menu

```
rm trans_menu.c
cp trans_menu.body trans_menu.c
echo 'cc -o dm_menu trans_menu.c'
cc -o dm_menu trans_menu.c
```

Data Dictionary-Data Manager File Translator

The DD-DM File Translator program translates a Standard Data File into a Data Dictionary Editor File format and vice-versa.

Include Files:

dmdefine.h	define.h (same as DM)

Source:

DM TO DD CODE	DD TO DM CODE
dmtodd.c	ddtodm.c
dmtoddlib.c	ddtodmlib.c
build_dmtodd_req.c	build_ddtodm_req.c
build_dmtodd_design.c	build_ddtodm_design.c

DM to DD Makefile

```
#
# Makefile for dmtodd
#

dmtodd : dmtodd.o build_dmtodd_req.o build_dmtodd_design.o \
    dmtoddlib.o
    ld -o dmtodd /lib/crt0.o dmtodd.o build_dmtodd_req.o \
        build_dmtodd_design.o dmtoddlib.o -lc

dmtodd.o : dmtodd.c dmdefine.h define.h
      cc -c dmtodd.c

build_dmtodd_design.o : build_dmtodd_design.c dmdefine.h define.h
                  cc -c build_dmtodd_design.c

dmtoddlib.o : dmtoddlib.c dmdefine.h define.h
          cc -c dmtoddlib.c
```

DD to DM Makefile

```
#
# Makefile for ddtodm
#

ddtodm : ddtodm.o build_ddtodm_req.o build_ddtodm_design.o \
    ddtodmlib.o
    ld -o ddtodm /lib/crt0.o ddtodm.o build_ddtodm_req.o \
        build_ddtodm_design.o ddtodmlib.o -lc

ddtodm.o : ddtodm.c dmdefine.h define.h
       cc -c ddtodm.c

build_ddtodm_design.o : build_ddtodm_design.c dmdefine.h define.h
                    cc -c build_ddtodm_design.c

ddtodmlib.o : ddtodmlib.c dmdefine.h define.h
          cc -c ddtodmlib.c
```

Attachment 1

Data Manager Makefile

The following makefile (DM_Makefile) is used to generate the DM executable code. It is called with the make.dm script.

```
#
# Makefile for dm.exe
#

dm.exe : dm_main.o trans_build_edit.o build_parent_list.o \
rmain.o session_ret_hdr_mod.o retrieve_driver_mod.o class_1_ret.o \
class_2_ret.o class_3_ret.o class_4_ret.o class_5_ret.o class_6_-
ret.o\
class_8_ret.o \
wmain.o session_wrt_hdr_mod.o write_driver_mod.o class_1_write.o \
class_2_write.o class_3_write.o class_4_write.o class_5_write.o \
class_6_write.o class_7_write.o class_8_write.o \
error_routines.o proglib.o
  ld -o dm.exe /lib/crt0.o dm_main.o trans_build_edit.o build_pare-
nt_list.o \
rmain.o session_ret_hdr_mod.o retrieve_driver_mod.o class_?_ret.o \
wmain.o session_wrt_hdr_mod.o write_driver_mod.o class_?_write.o \
error_routines.o proglib.o -lq -lc

dm_main.o : dm_main.body define.h datadef.h ingdata.h
  - rm dm_main.q
  - rm dm_main.c
  cat ingdata.h dm_main.body >> dm_main.q
  equel dm_main.q
  cc -c dm_main.c

error_routines.o : error_routines.body define.h datadef.h ingdata.h
  - rm error_routines.q
  - rm error_routines.c
  cat ingdata.h error_routines.body >> error_routines.q
  equel error_routines.q
  cc -c error_routines.c

build_parent_list.o : build_parent_list.body define.h datadef.h in-
gdata.h
  - rm build_parent_list.q
  - rm build_parent_list.c
  cat ingdata.h build_parent_list.body >> build_parent_list.q
  equel build_parent_list.q
  cc -c build_parent_list.c
```

```
proglib.o : proglib.body define.h datadef.h ingdata.h
  - rm proglib.q
  - rm proglib.c
  cat ingdata.h proglib.body >> proglib.q
  equel proglib.q
  cc -c proglib.c

class_1_ret.o : class_1_ret.body define.h datadef.h ingdata.h
  - rm class_1_ret.q
  - rm class_1_ret.c
  cat ingdata.h class_1_ret.body >> class_1_ret.q
  equel class_1_ret.q
  cc -c class_1_ret.c

class_2_ret.o : class_2_ret.body define.h datadef.h ingdata.h
  - rm class_2_ret.q
  - rm class_2_ret.c
  cat ingdata.h class_2_ret.body >> class_2_ret.q
  equel class_2_ret.q
  cc -c class_2_ret.c

class_3_ret.o : class_3_ret.body define.h datadef.h ingdata.h
  - rm class_3_ret.q
  - rm class_3_ret.c
  cat ingdata.h class_3_ret.body >> class_3_ret.q
  equel class_3_ret.q
  cc -c class_3_ret.c

class_4_ret.o : class_4_ret.body define.h datadef.h ingdata.h
  - rm class_4_ret.q
  - rm class_4_ret.c
  cat ingdata.h class_4_ret.body >> class_4_ret.q
  equel class_4_ret.q
  cc -c class_4_ret.c

class_5_ret.o : class_5_ret.body define.h datadef.h ingdata.h
  - rm class_5_ret.q
  - rm class_5_ret.c
  cat ingdata.h class_5_ret.body >> class_5_ret.q
  equel class_5_ret.q
  cc -c class_5_ret.c

class_6_ret.o : class_6_ret.body define.h datadef.h ingdata.h
  - rm class_6_ret.q
  - rm class_6_ret.c
  cat ingdata.h class_6_ret.body >> class_6_ret.q
  equel class_6_ret.q
  cc -c class_6_ret.c
```

```
class_8_ret.o : class_8_ret.body define.h datadef.h ingdata.h
  - rm class_8_ret.q
  - rm class_8_ret.c
  cat ingdata.h class_8_ret.body >> class_8_ret.q
  equel class_8_ret.q
  cc -c class_8_ret.c

retrieve_driver_mod.o : retrieve_driver_mod.body define.h datadef.h
ingdata.h
  - rm retrieve_driver_mod.q
  - rm retrieve_driver_mod.c
  cat ingdata.h retrieve_driver_mod.body >> retrieve_driver_mod.q
  equel retrieve_driver_mod.q
  cc -c retrieve_driver_mod.c

rmain.o : rmain.body define.h datadef.h ingdata.h
  - rm rmain.q
  - rm rmain.c
  cat ingdata.h rmain.body >> rmain.q
  equel rmain.q
  cc -c rmain.c

session_ret_hdr_mod.o : session_ret_hdr_mod.body define.h datadef.h
ingdata.h
  - rm session_ret_hdr_mod.q
  - rm session_ret_hdr_mod.c
  cat ingdata.h session_ret_hdr_mod.body >> session_ret_hdr_mod.q
  equel session_ret_hdr_mod.q
  cc -c session_ret_hdr_mod.c

session_wrt_hdr_mod.o : session_wrt_hdr_mod.body define.h datadef.h
ingdata.h
  - rm session_wrt_hdr_mod.q
  - rm session_wrt_hdr_mod.c
  cat ingdata.h session_wrt_hdr_mod.body >> session_wrt_hdr_mod.q
  equel session_wrt_hdr_mod.q
  cc -c session_wrt_hdr_mod.c

trans_build_edit.o : trans_build_edit.body define.h datadef.h
ingdata.h
  - rm trans_build_edit.q
  - rm trans_build_edit.c
  cat ingdata.h trans_build_edit.body >> trans_build_edit.q
  equel trans_build_edit.q
  cc -c trans_build_edit.c
```

```
class_1_write.o : class_1_write.body define.h datadef.h ingdata.h
  - rm class_1_write.q
  - rm class_1_write.c
  cat ingdata.h class_1_write.body >> class_1_write.q
  equel class_1_write.q
  cc -c class_1_write.c

class_2_write.o : class_2_write.body define.h datadef.h ingdata.h
  - rm class_2_write.q
  - rm class_2_write.c
  cat ingdata.h class_2_write.body >> class_2_write.q
  equel class_2_write.q
  cc -c class_2_write.c

class_3_write.o : class_3_write.body define.h datadef.h ingdata.h
  - rm class_3_write.q
  - rm class_3_write.c
  cat ingdata.h class_3_write.body >> class_3_write.q
  equel class_3_write.q
  cc -c class_3_write.c

class_4_write.o : class_4_write.body define.h datadef.h ingdata.h
  - rm class_4_write.q
  - rm class_4_write.c
  cat ingdata.h class_4_write.body >> class_4_write.q
  equel class_4_write.q
  cc -c class_4_write.c

class_5_write.o : class_5_write.body define.h datadef.h ingdata.h
  - rm class_5_write.q
  - rm class_5_write.c
  cat ingdata.h class_5_write.body >> class_5_write.q
  equel class_5_write.q
  cc -c class_5_write.c

class_6_write.o : class_6_write.body define.h datadef.h ingdata.h
  - rm class_6_write.q
  - rm class_6_write.c
  cat ingdata.h class_6_write.body >> class_6_write.q
  equel class_6_write.q
  cc -c class_6_write.c

class_7_write.o : class_7_write.body define.h datadef.h ingdata.h
  - rm class_7_write.q
  - rm class_7_write.c
  cat ingdata.h class_7_write.body >> class_7_write.q
  equel class_7_write.q
  cc -c class_7_write.c
```

```
class_8_write.o : class_8_write.body define.h datadef.h ingdata.h
  - rm class_8_write.q
  - rm class_8_write.c
  cat ingdata.h class_8_write.body >> class_8_write.q
  equel class_8_write.q
  cc -c class_8_write.c


wmain.o : wmain.body define.h datadef.h ingdata.h
  - rm wmain.q
  - rm wmain.c
  cat ingdata.h wmain.body >> wmain.q
  equel wmain.q
  cc -c wmain.c

write_driver_mod.o : write_driver_mod.body define.h datadef.h
ingdata.h
  - rm write_driver_mod.q
  - rm write_driver_mod.c
  cat ingdata.h write_driver_mod.body >> write_driver_mod.q
  equel write_driver_mod.q
  cc -c write_driver_mod.c
```

Appendix G: Summary Paper

Abstract

This paper describes the design and implementation of a common database interface which integrates a set of heterogeneous software engineering tools. These tools run on a variety of workstations and are combined to form System 690 which provides a software design environment for use within the Air Force Institute of Technology (AFIT) Software Engineering Laboratory (SEL). The interface was implemented using a standard data file for all data transfer and a data manager which provides the database support for the System 690 tools. The unique aspects of the interface are its ability to support tool data changes and the ability to incorporate new tools into System 690.

Introduction

The goal of System 690 is to provide an integrated system in which a designer could sit down at a workstation, download the necessary data from a central database, work on a portion of the design, and when finished, upload the modified data back to the database. This data, when stored in a comprehensive, centralized database, would provide a system which could share data between tools and provide the means to document a software project throughout its entire life cycle.

The objective of this research was to integrate the System 690 tools by designing and implementing a common database interface between the tools and a central database. The interface was implemented using a standard data file to transfer data between the tools and a data manager which performs all database transactions. The primary design consideration was for the interface to support the incorporation of new tools into System 690.

Overall System Analysis

The basic objective of System 690 is to support the standard software development methodology established in the Software Development Documentation Guidelines and Standards (5). These guidelines establish the software development documentation standards for all AFIT software development projects. The method used to support this standard is a data dictionary. A dictionary entry is established for the requirements, design, and implementation phases of the software life cycle. Each of these phases consists of a set of action entities and a set of object entities for a total of six types of dictionary entries. Refer to Figure 1 for a sample data dictionary entry.

Several thesis efforts have produced a set of automated tools and a data dictionary database which support the concepts set forth in the Software Development Documentation Guidelines and Standards (5). The data dictionary database contains the schema for all six data dictionary entries.

```
NAME: mess_parts
PROJECT: NETOS-ISO
TYPE: PARAMETER
DESCRIPTION: Decomposed message parameters.
DATA TYPE: Composite, probably C structure or PASCAL record.
MIN VALUE: None
MAX VALUE: None
RANGE OF VALUES: None
VALUES: None
PART OF: None
COMPOSITION:    SRC
                DST
                SPN
                DPN
                USE
                QTY
                Buffer
ALIAS: Message Parts
   WHERE USED: Passed from Decompose Message to Validate Parts
   COMMENT: Part of earlier design
ALIAS: messy-parts
   WHERE USED: Passed from Dump Data to Flush Buffer.
   COMMENT: Part of existing library.
REFERENCE: MSG_PARTS
   REFERENCE TYPE: SADT
VERSION: 1.2
VERSION CHANGES: Component USE added to allow network messages
DATE: 11/05/85
AUTHOR: T. C. Hartrum
CALLING PROCESS: Process Message
   PROCESS CALLED: Decompose_Message(parts list)
   DIRECTION: up
   I/O PARAMETER NAME: parts_list
CALLING PROCESS: Process Message
   PROCESS CALLED: Process Network 4 Messages
   DIRECTION: down
   I/O PARAMETER NAME: parts
```

Figure 1. Sample Object Entity Dictionary Entry in Design Phase (5: 29)

Refer to Figure 2 for the schema of the object entity in the design phase. The data dictionary database was implemented

using the Ingres relational DBMS and runs under the Unix operating system on a VAX 11/785.

parameter		papassed	
project	c12	project	c12
paname	c25	paname	c25
datatype	c25	prcalling	c25
low	c15	prcalled	c25
hi	c15	direction	c4
span	c60	iopaname	c25
status	c1		
padesc		**pavalueset**	
project	c12	project	c12
paname	c25	paname	c25
line	i2	value	c15
description	c60		
paalias		**pahierarchy**	
project	c12	project	c12
paname	c25	hipaname	c25
aliasname	c25	lopaname	c25
comment	c60		
whereused	c25		
pahistory		**paref**	
project	c12	project	c12
paname	c25	paname	c25
version	c10	reference	c60
date	c8	reftype	c25
author	c20		
comment	c60		

Figure 2. Database Schema for an Object Entity Within the Design Phase (4: 37)

The existing workstation tools are connected to a VAX 11/785 via a Gandalf network, which creates an excellent opportunity to create an integrated environment where all the tools share data using the Ingres DBMS. However, prior to this research, only the design phase of the tools could

interface with Ingres. This configuration (see Fig. 3) prevented the tools from sharing information and precluded the use of tools such as automated consistency checkers, which could provide design consistency throughout the various phases of a system's design (6: 652). The inability of the tools to use a common database was the main problem which prevented integrating the System 690 tools and was the basis for performing this research.

System Design Analysis

Before beginning the design analysis, several definitions are needed. A data entity refers to all the information describing a data dictionary entry. The data entity consists of multiple data elements. These data elements are the values representing specific data fields in a data dictionary entry. A session refers to a tool-data manager interaction where data is retrieved from the database, manipulated by the tool, and stored back into the database. A transaction is a request to the data manager to perform a database retrieval or update.

The primary requirements of the data manager and standard data file were for them to adapt to tool data changes and support the addition of new tools with their different file requirements. The data manager performs many tasks, but its basic function is to retrieve data from and write data to a common database using the standard data file. These components will interact in every database

transaction (see Fig. 4). The following sample session is provided to show how the data manager and standard data file will interact to update the common database.

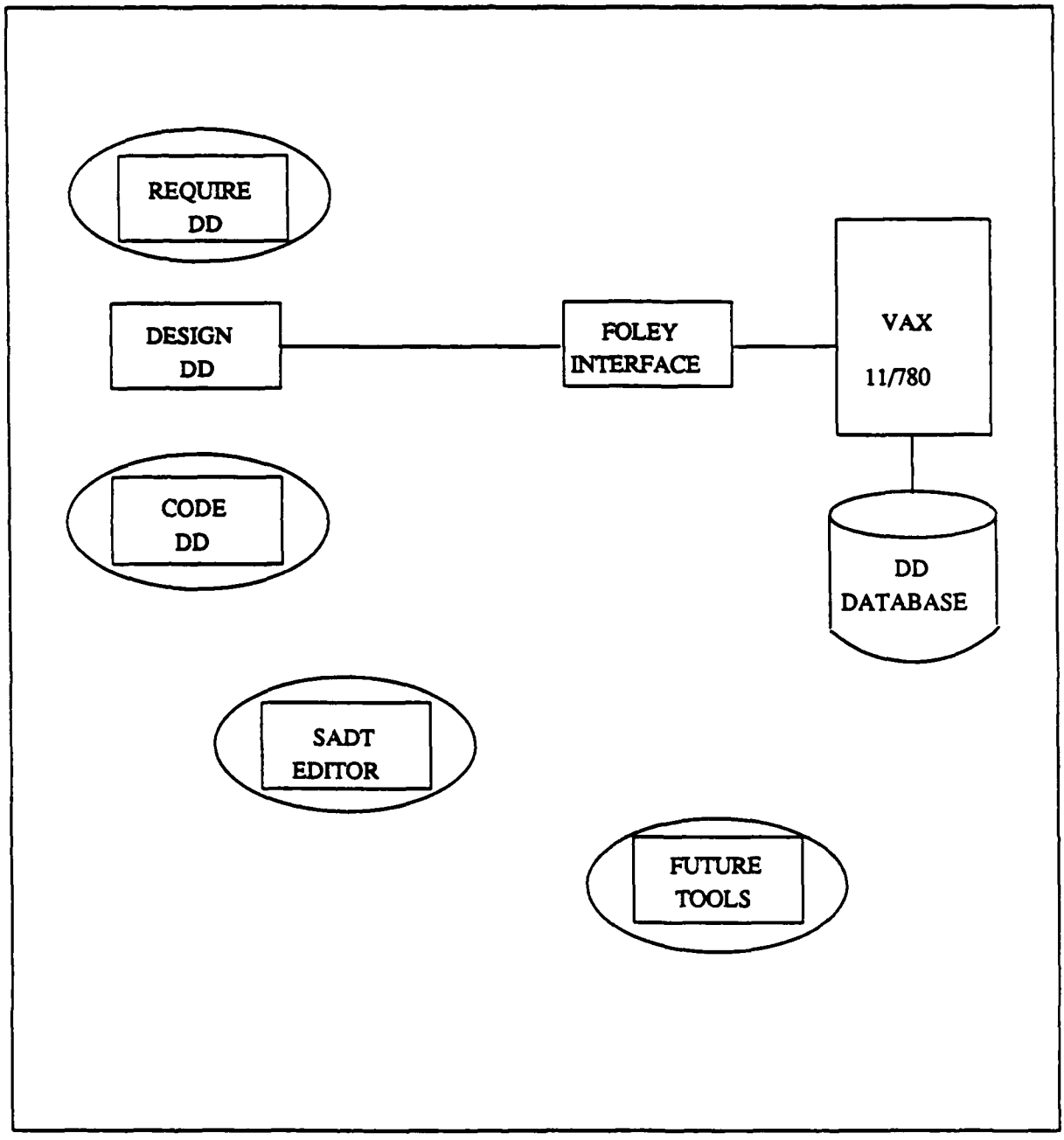


Figure 3. Current System 690 Configuration

Sample Session: A user or tool will request a data entity(s) from the database. On receipt of the request, the data manager will retrieve the data and provide this data back to the requestor in a standard data file. When retrieving the data, the data manager will provide session control to maintain database integrity.

When the desired changes have been made to the data, the tool which checked-out the data, requests to update the database. The data manager will use the standard data file, containing any changes made by the tool, and the session information generated during the retrieval to coordinate and perform the database updates. After the data is successfully written back to the database, the session is terminated.

The sample session shows how a typical session is performed. It also indicates the dual role the standard data file and session information provide. The impact of this dual role will be seen throughout the remainder of this paper.

Standard Data File. The standard data file is the means used by the data manager to transfer data between System 690 tools and the common database. It provides a standard file structure for all tools to use in interfacing with the data manager. The file is the interface and therefore must contain not only the requested tool data, but also provide control information to the tool and data manager by describing the contents of the file.

The design of the standard file is examined with respect to its two components: file description header and data file entries. The file description header design is based on the need for it to provide the information neces-

sary to inform a tool and the data manager what the contents and structure of the file are. The data file entries' format design is based on identifying the types and structures of the data elements being transferred.

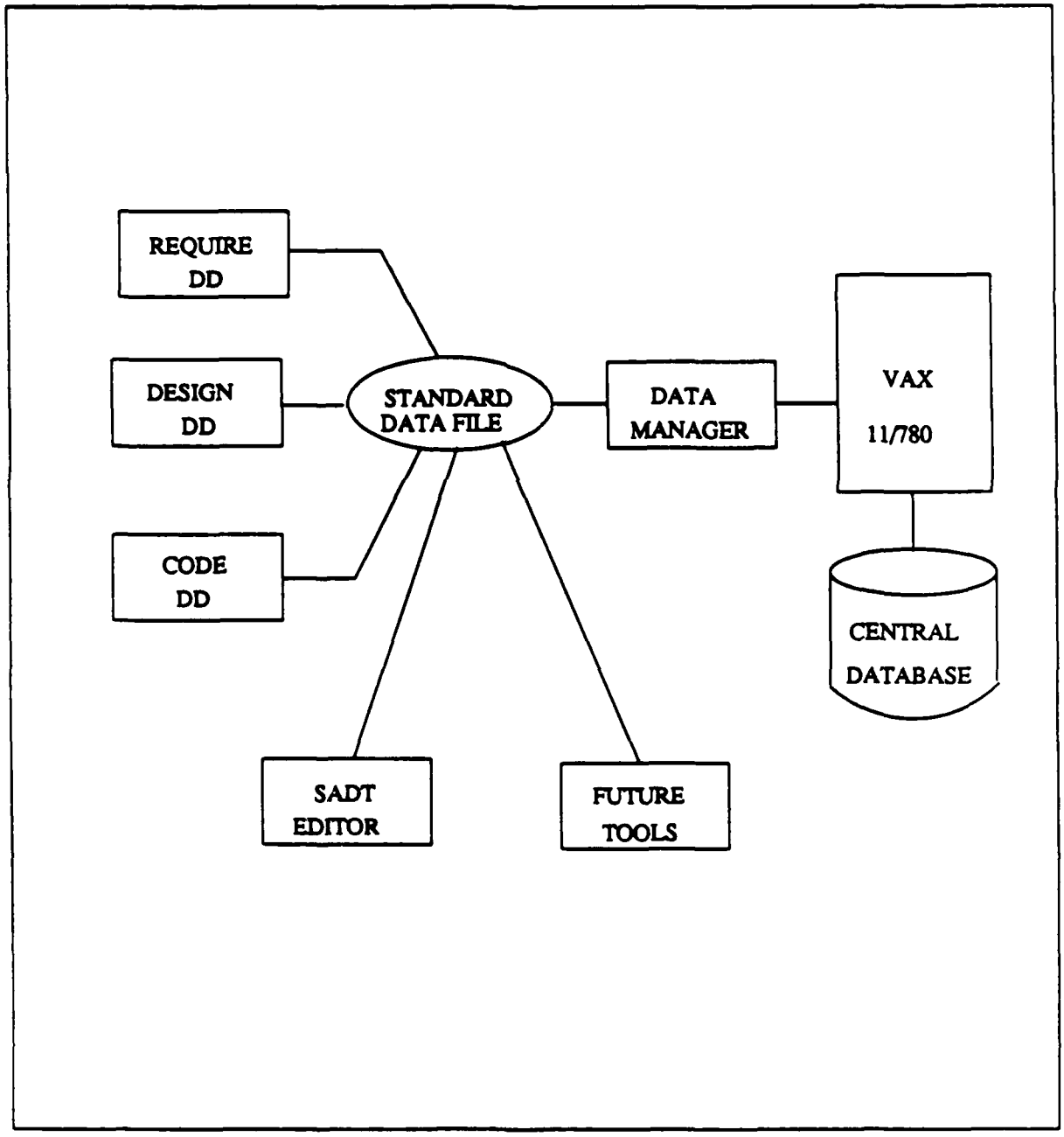


Figure 4. Requirements System 690 Configuration

File Description Header. The contents of the file description header are the following: session identification, tool/file compatibility header, project, phase, type, data entity summary, and start/stop time entries. These fields were combined to produce the file description header (Fig. 5) used in the standard data file.

The file description header supports the standard data file's dual role of describing the type of data contained in the file to both the tools and the data manager. The session identification field is used to support the data manager in performing its session control function. The project, phase, and type identify the specific type and format of the data entities contained in the data portion of the file. The entity list provides the names and types of the entities in the file. This allows the data manager to verify that the file's data contents correspond to the header entity list to insure that no entities have been erroneously added to or deleted from the data file.

Data File Entries. The data portion of the standard data file consists of one or more data entity entries. Each entry is composed of all the data elements necessary to satisfy a data dictionary entry. The data elements are contained in a series of data records (Fig. 6) and consist of the following fields: dataname, field length, multi-line indicator, number of fields, direction, type, and contents.

These fields provide a full description of a data element and its use in a data dictionary field. The ability to describe a data element allows the element records in a file to be ordered to satisfy a tool's specific data requirements.

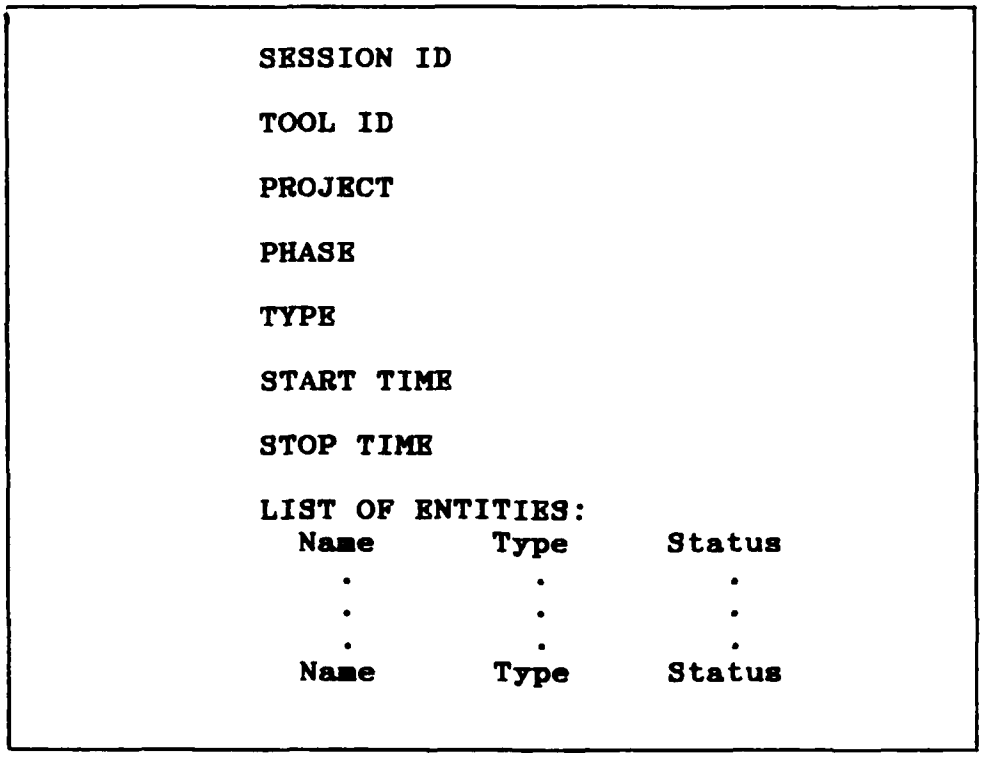


Figure 5. File Description Header Format

Data File Structure. The standard data file structure (Fig. 7) is built using the file description header and data entity entries. The file contains all ASCII characters and consists of the file description header, data entities, and section delimiters. The delimiters are unique for each section and are designed to help the tools and data manager maintain their position in the file. The delimiters

also help tool developers read the file's contents for debugging purposes.

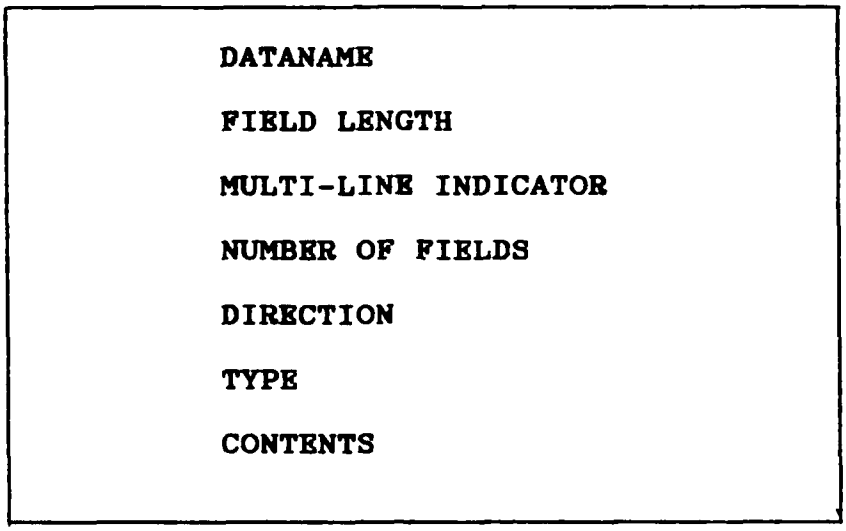


Figure 6. Data Element Record Format

The key feature of the data file is the ability to place the data element records in a tool-specified order. The file description header contains the information describing this ordering to both the tool and the data manager. The capability to support multiple data record orderings allows the standard data file to incorporate tool data changes and to add new tools to System 690.

Data Manager. The data manager must provide a broad range of functions. Its primary functions are to retrieve data from and write data to the database using the standard data file. The data manager also provides an interface which allows tools and users to specify the transactions to

be performed. Finally, it provides session control to protect database integrity.

```
#@@BEGIN@@#
#@#HEADER BEGIN#@#

<file description header, Fig. 5>

#@#HEADER END#@#
###ACTION TYPE###

@##START##@

<entity element record, Fig. 6>

@##STOP##@

    o
    o
    o

###ACTION END###

###OBJECT TYPE###

@##START##@

<entity element record, Fig. 6>

@##STOP##@

    o
    o
    o

###OBJECT END###

#@@END@@#
```

Figure 7. Standard Data File Format

Database Functions. The primary components used by the data manager to perform database transactions are the tool data definition table and the tool description table. These tables permit the generic classification of the entities used by a tool. They support the data manager's two primary functions which are to perform the database retrievals necessary to generate the standard data file and to use the standard data file to perform database updates.

Tool Data Definition Table. The key data manager requirements are for it to support the retrieval of data dictionary entries, formatting the retrieved data into the standard file format, and updating the database. The data definition table (Fig. 8) provides a mechanism which is flexible enough to incorporate current and future tool data requirements into a standard data file with little or no data manager programming being required. The table provides all the information necessary to retrieve or update a data element and it contains the information necessary to read and write the standard data file.

A data definition table is created for each data entity type used by a tool. Each table has a unique relation name and is tool, phase, and type specific. The use of multiple relations localize the impact of tool changes and supports the requirement to easily incorporate new tools into System 690. To add a new tool, the only requirement is for the appropriate tool data definition table(s) be created.

The key fields in the table are the data element name, element's relation name, relation's key names, and the entry classification of the element's relation. The data name, relation name, and key fields contain the information necessary to identify any data field in the database. The entry class identifies the structure of the data element and its access method. The combination of the these fields allows the data manager to access and modify any database data element.

DATANAME	RELATION	KEYFIELD_1	KEYFIELD_2

FIELD_DESCRIPTION	ENTRY_CLASS	MULTI_LINE_INDICATOR

NUMBER_OF_FIELDS	DIRECTION	TYPE

DELETE_FLAG	VERSION	LINE

Figure 8. Tool Data Definition Table

The ability to use these fields to update and retrieve any database element is the key feature of the data manager. The data definition table entries may be in any order. This ordering dictates the order the data elements are written and read from the standard data file. Because the fields

may be placed in an arbitrary order, the data manager can support any ordering of data and thus can support a tool's specific file requirements allowing for the easy addition of new tools to System 690.

The data definition table not only supports the easy incorporation of new tools, but it does so in a generic manner. The only requirement to support a new tool is to identify the entry classification of its data elements. If the entry classes are the same as existing classes no programming changes must be made to the data manager. If no class exists for a specific data element, only changes to support this new class is required. This greatly enhances the data manager's ease-of-use for tool designers and it reduces its maintenance to a very minimal level.

Tool Description Table. The tool description table (Fig. 9) describes a tool and its data needs to the data manager. The description table is used by the data manager for transaction request verification and database retrievals and updates. There is a tool description table entry for each phase and type of data entity used by a tool. This is required to identify the specific data definition table relation describing the requested data dictionary entry. This table, in conjunction with the data definition table, allows the data manager to identify the standard data file requirements for any System 690 tool. The description table identifies the data definition table to use in reading

or writing the standard data file and the data definition table contains the file's structure.

TOOL_NAME	PHASE	TYPE	DEFINITION_TABLE	DESCRIPTION

Figure 9. Tool Description Table

Tool/User Interface. The tool/user interface is provided to allow a tool/user to specify to the data manager the type of transaction to perform and provide the tool/user the transaction's results. The interface supports both interactive and batch requests. This provides greater flexibility for smart tools that can build batch requests without the user having to interface directly with the data manager (Fig. 10). The interactive interface (Fig. 11) is available for use with tools that do not have the sophistication to perform a batch transaction.

Tool Data Request. The tool data request (Fig. 12) contains the information needed by the data manager to perform all of its database transactions. The majority of the tool data request correspond to those used in the session file description header. The fields unique to the request file are the transaction indicator and the parent and levels entries.

The transaction indicator informs the data manager the action it is to take. The transactions supported by the

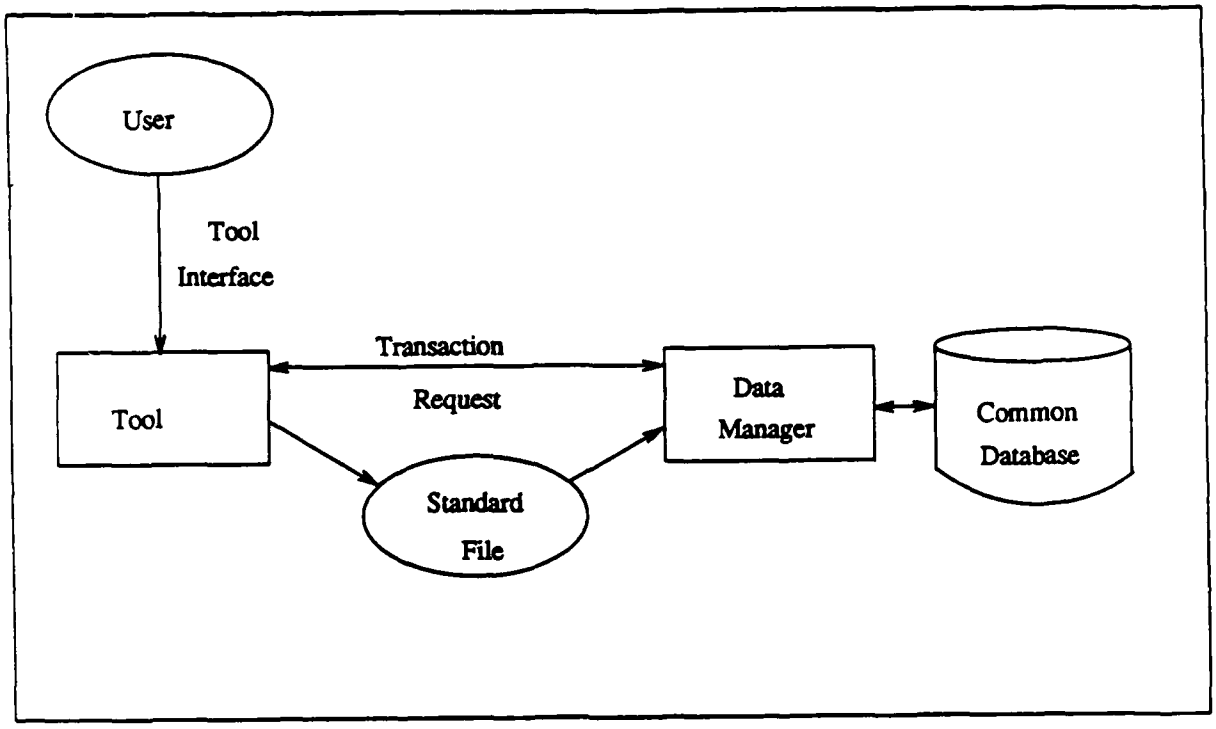


Figure 10. Batch Data Manager Interface

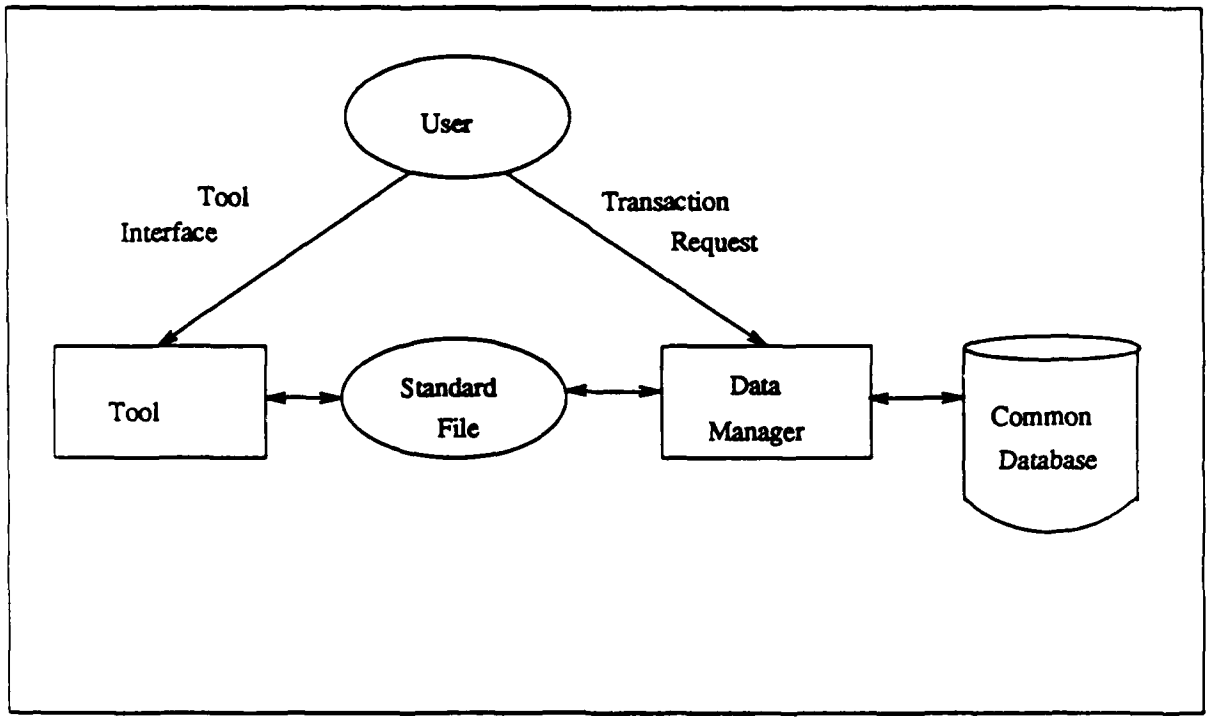


Figure 11. Interactive Data Manager Interface

data manager are data retrievals, updates, deletions, and session aborts. The data delete function is provided to allow a user to delete specified entities without having to retrieve them for update, changing their status to delete, and resubmitting them for write with update. The session abort transaction was added to provide an easy means for users or the database administrator to abort an old or corrupted session. This allows data entities which had been identified as checked-out for use to be made available for other users.

The parent and levels fields are provided to allow a user an easy means of retrieving a large set of related data entities by providing a single parent name and the entities pointed to by that parent for the specified number of levels. An example of this is an SADT diagram which contains multiple action and object entities all of which are pointed to by its Title. This is an important feature because new software design tools are incorporating the ability to simultaneously work with multiple levels of a system's design. This feature precludes having the user or tool track the entities needed for a session and eliminates the possibility of omitting entities needed within a session.

Results Reporting. All transaction results are reported back to the tool/user. Batch transaction results are reported through the use of a results file. The results

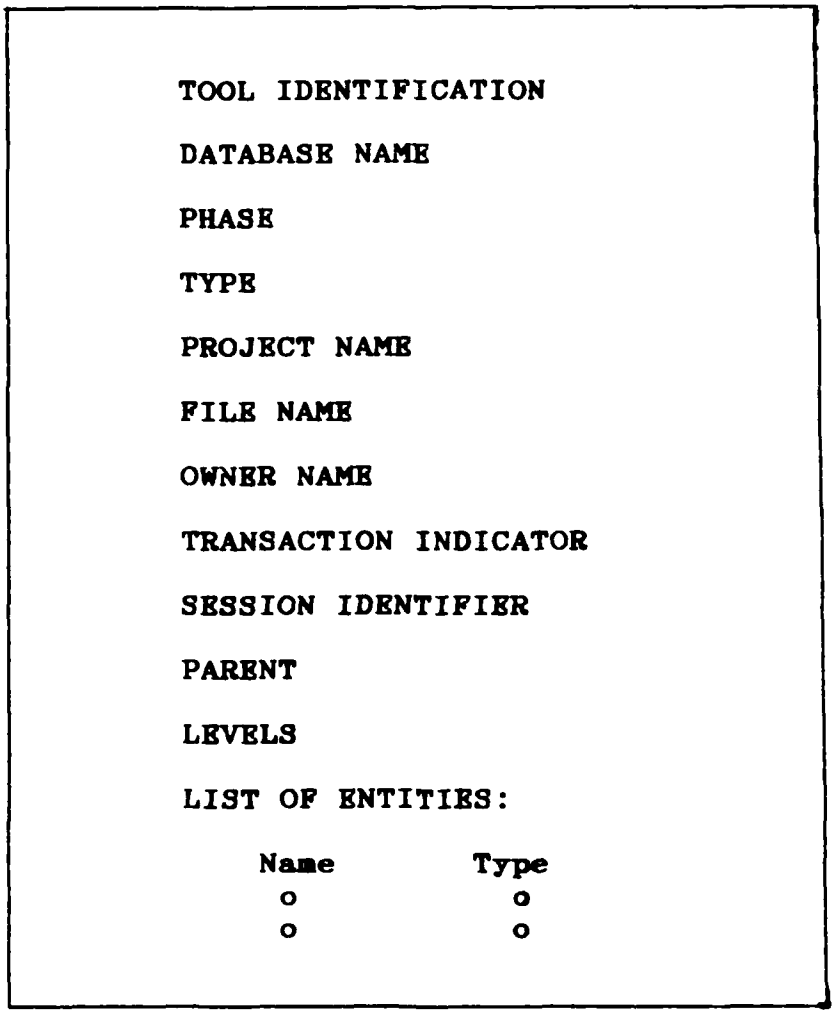


Figure 12. Tool Data Request Format

file contains the list of successfully performed transactions. In the case of an error, the cause and error recovery results are placed in the results file. For interactive transactions, the same results are reported to the user but the results are displayed directly to the user via the CRT screen.

Session Control. Session control provides the data manager librarian function. During retrievals, the data

manager determines the status of all requested data entities and generates the session control information. During updates, the data manager uses this information to perform transaction verification to insure that only the permitted modifications have been requested. This session information is maintained in two tables: session entity table and session identification table.

Session Entity Table. The session entity table (Fig. 13) tracks each entity used in a session, its type, and update status. The session id corresponds to the associated session identifier. The session identifier uniquely identifies the entities in a session. The identifier is used by the data manager to check data entities back into the database. The status field indicates whether the entity is in a Read or Write status. Only those entities in a Write status may be updated during a session.

SESSION_ID	NAME	TYPE	STATUS

Figure 13. Session Entity Table

Session Identification Table. The session identification table (Fig. 14) maintains the status of a session, describes the type of data used in a session, and identifies the session's owner and tool being used. This table supports the data manager update function in verifying session update requests. It also provides the database

administrator an easy means to identify session owners. This benefit of this is the ability to contact a user to check a session file in when another user needs the same data entities for update purposes.

PROJECT	PARENT_NAME	LEVELS	PHASE	TYPE

SESSION_ID	OWNER	TOOL

Figure 14. Session Identification Table

Common Database

The basic design of the common database is well documented in Thomas' thesis (16: 84-142). The data manager required few extensions to this design. These extensions are required to support the various tables used and to enable entity status tracking.

Thomas intended for only one database to be used for all three data dictionary phases. However for flexibility, this is no longer the case. A database may now contain one or more phases. The only requirement is both entity types (action and object) used within a phase be present.

The data manager's support of multiple database provides the opportunity to split the database across systems. This is important because it provides easier

access for tools and reduces the processing load on the various systems. This is especially important within the AFIT environment where the computer systems suffer severe performance degradation during certain periods of the school year.

Implementation

The computer resources available within the SEL dictated the configuration (Fig. 15) used to implement the data manager, central database, and standard data file. The data manager was implemented on a VAX 11/785 computer, using the Berkeley 4.3 Unix operating system. The central database was developed using the Ingres relational DBMS. The data manager was developed using the "C" programming language. The queries were performed using the Embedded Query Language (EQUEL) provided with Ingres.

To evaluate the data manager and standard data file implementation, two tools, a data dictionary editor and an enhanced SADT editor were modified to interface with the standard data file. Both tools were able to successfully use the common interface and access the central database. The most important result of this integration was the ease in which the tools were modified to support the interface. The programmers modifying the tools found the standard data file to support the integration very well and did not require extensive programming effort to incorporate its use with their tools.

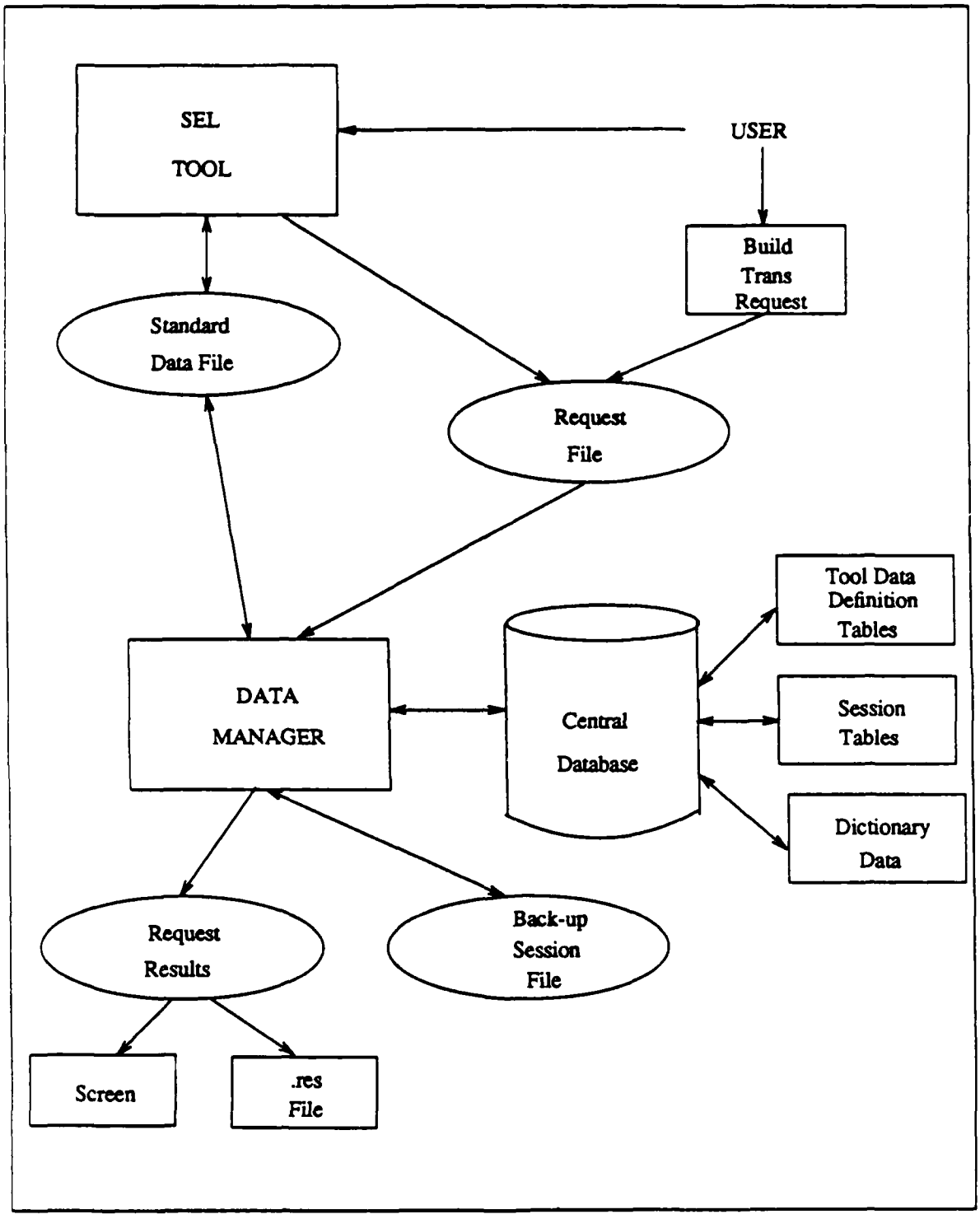


Figure 15. Overall System Implementation

Summary

The objective of this research was to implement a common database interface which integrated the SEL tools to form System 690. The key design consideration was for the interface to support not only the existing tools but also support the addition of new tools.

The interface was implemented using a standard data file and a data manager. The key feature of these two components is their ability to support multiple file configurations without requiring programming changes to the tools or the data manager.

The common database interface successfully integrated all the tools currently within the SEL to form a fully integrated System 690. The interface easily supported the integration of the tools without requiring an extensive coding effort for either the tools or the data manager.

The benefit of this interface is just being seen. Previously, the difficulty of designing a tool and trying to develop a database interface was too overwhelming for a single researcher which has resulted in limited new tool development. Hopefully researchers, without the burden of developing a complete database interface for their tools, will be encouraged to develop new tools and with a higher degree of sophistication.

Bibliography

1) Barabino, G. P. and others. "A Module for Improving Data Access and Management in an Integrated CAD Environment," Proceedings of the IEEE Twenty-Second Design Automation Conference. 577-583. Silver Spring, MD: IEEE Computer Society Press, 1985.

2) -----. "A Modular System for Data Management in VLSI Design," Proceedings of the ACM/IEEE International Conference on Computer Design. 796-801. Silver Spring, MD: IEEE Computer Society Press, 1984.

3) Fedchak, Elaine. "An Introduction to Software Engineering Environments," Proceedings of the IEEE Tenth International Computer Software and Applications Conference. 456-463. Washington D.C.: IEEE Computer Society Press, 1986.

4) Foley, Capt Jeffrey W. Design of a Data Dictionary Editor in a Distributed Software Development Environment. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1986 (AD-A152406).

5) Hartrum, Thomas C. Software Development Documentation Guidelines and Standards (Draft 3a). School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1986.

6) Hartrum, Thomas C. and Capt Charles W. Hamberger. "Development of a Distributed Data Dictionary System for Software Development," Proceedings of the IEEE 1986 National Aerospace and Electronics Conference, 3:648-655. New York: IEEE Press, 1986.

7) Horowitz, Ellis and Ronald Williamson. "SODOS: A Software Documentation Support Environment: Its Use," Proceedings of the IEEE Eighth International Conference on Software Engineering. 8-14. Silver Spring, MD: IEEE Computer Society Press, 1985.

8) Hsu, Arding and others. "A Design Environment That Integrates Tools, Database, and User Interface," Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers. 733-736. Silver Spring, MD: IEEE Computer Society Press, 1984.

9) Johnson, Capt Steven E. A Graphics Editor for Structured Analysis with a Data Dictionary. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

10) Kalay, Yehunda E. "A Database Management Approach to CAD/CAM Systems Integration," Proceedings of the IEEE Twenty-Second Design Automation Conference. 111-116. Silver Spring, MD: IEEE Computer Society Press, 1985.

11) Katz, Randy H. "Managing the Chip Design Database," IEEE Computer, 16: 26-36 (December 1983).

12) Katz, Randy H. and Tobin J. Lehman "Database Support for Versions and Alternatives of Large Design Files," IEEE Transactions on Software Engineering, 10: 191-200 (March 1984).

13) Pressman, Roger S. Software Engineering: A Practitioner's Approach (Second Edition). New York: McGraw-Hill Book Company, 1987.

14) Purtilo, James. "Polylith: An Environment to Support Management of Tool Interfaces," Papers of ACM SIGPLAN 85 Symposium. 12-18. New York: Association of Computing Machinery, 1985.

15) Stucki, Leon G. "What About CAD/CAM for Software? The ARGUS Concept," Proceedings of the IEEE Conference on Software Development Tools, Techniques, and Alternatives. 129-135. Silver Spring, MD: IEEE Computer Society Press, 1983.

16) Thomas, Capt Charles W. An Automated/Interactive Software Engineering Tool to Generate Data Dictionaries. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984 (AD-A152215).

17) Urscheler, Capt James. An Interactive Graphics Editor for SADT Diagrams. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986 (AD-A177663).

VITA

Captain Ted D. Connally was born on 4 August 1958 in Stamford, Texas. He graduated from Stamford High School in 1976 and attended Texas A&M University, from which he received the degree of Bachelor of Science in Computer Science in May 1980. Upon graduation, he received a commission in the USAF through the ROTC program. He entered active duty in June 1980 at Randolph AFB, Texas where he served as Programming Team Chief, 3302nd Computer Services Squadron until July 1984. He then served as Chief, Information Systems Branch, Air Force Coordinating Office for Logistics Research, Wright-Patterson AFB, Ohio, until entering the School of Engineering, Air Force Institute of Technology in May 1986.

Permanent Address: 603 Dodson Drive
Stamford, Texas 79553

END

DATE

3-88

DTIC